Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

DSLTranslator - Ferramenta para Transformação de Modelos

Roberto Félix (aluno nº 29405)

1º Semestre de 2009/10

22 de Fevereiro de 2010

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

DSLTranslator - Ferramenta para Transformação de Modelos

Roberto Félix (aluno nº 29405)

Orientador: Prof. Doutor Vasco Amaral

Trabalho apresentado no âmbito do Mestrado em Engenharia Informática, como requisito parcial para obtenção do grau de Mestre em Engenharia Informática.

1º Semestre de 2009/10
22 de Fevereiro de 2010

Agradecimentos

Gostaria em primeiro lugar de agradecer ao Professor Vasco Amaral pela possibilidade de trabalhar num projecto ambicioso, que valoriza todo o trabalho efectuado, e também pela orientação fornecida durante toda a duração desta dissertação.

Agradeço também:

- À equipa do grupo SOLAR, nomeadamente ao Bruno Barroca pela ajuda intensiva durante toda a implementação do projecto e pela quantidade e qualidade de ideias fornecidas para o desenvolvimento deste, ao Doutor Levi Lúcio pela ajuda na formalização da semântica da linguagem de transformação, ao Vasco Sousa pela ajuda na construção do editor da linguagem que é parte fundamental para um dos objectivos pretendidos neste trabalho e aos colegas Pedro Patrício, Rui Chambel, André Rosa, Ricardo Mateus e Ricardo Guerreiro pela disponibilidade de fornecer opiniões construtivas durante as reuniões de acompanhamento da dissertação.

- Ao projecto BATIC3S que juntamente com a Fundação para a Ciência e Tecnologia permitiram-me trabalhar nesta dissertação com bolsa de investigação

- A toda a minha família, em particular aos meus pais José e Luísa, e ao meu irmão Gonçalo, por me permitirem deixar a ilha para poder estudar na FCT/UNL sem que nunca me faltasse nada

- Aos amigos e colegas que de uma forma ou de outra contribuíram para o meu percurso académico culminado com esta dissertação de mestrado: Pedro Gabriel, Luís Silva, Nelson Fonte, Tiago Amorim, Flávio Martins, Rui Vieira, Marco Leitão, Ricardo Valladas, Pedro Pita, Guilherme Martins, Luís Assunção, Mário Cunha, Luís Oliveira, Hélio Pereira, Danilo Manmohanlal, João Mateus, António Costa...

A todos, por tudo,
Muito Obrigado!

Resumo

Com o aumento de soluções que facilitam a construção de linguagens de software, nomeadamente através de uma abordagem orientada a modelos, mas concentrados na definição da sintaxe concreta e abstracta, existe a necessidade de atribuição de significado a essas linguagens através da sua semântica.

É no seguimento deste objectivo que se pretende construir uma plataforma de prototipagem rápida de novas linguagens, nomeadamente possibilitando a atribuição rápida da sua semântica por transformação com o fim de definir uma linguagem em termos de uma outra formalmente bem definida e possivelmente de acordo com os conhecimentos do especialista da linguagem.

As diversas ferramentas actuais de transformação de modelos, não encontram um equilíbrio entre a usabilidade, validação e expressividade.

Neste trabalho, propomo-nos a implementar uma ferramenta de transformação baseada em gramáticas de grafos para definição e aplicação de regras sobre modelos e que consiga um equilíbrio adequado dos três critérios acima mencionados. Isto envolve desenhar a linguagem de especificação das regras de transformação, construção do editor visual e implementação da respectiva semântica.

Tendo em conta que esta linguagem, e correspondente editor, pretendem ser uma ajuda considerável ao desenvolvimento de novas linguagens, proporcionando aos especialistas da linguagem a atribuição de semântica através de transformações de modelos, a usabilidade terá, obrigatoriamente, que se centrar na simplicidade e numa curva de aprendizagem suave. Os utilizadores alvo podem assim desenvolver linguagens sem terem que se especializar numa plataforma de transformação e assim agilizar o processo de definição semântica de linguagens.

Palavras-chave: Transformações de modelos; Linguagens de Domínio Específico; Engenharia de Linguagens de Software; Desenvolvimento Orientado a Modelos; Transformações de Grafos;

Abstract

Currently, the offer of metamodeling frameworks allows an easy and rapid design of a software language, in particular through the Model Driven Development approach, but focused on the definition of concrete and abstract syntax, there is a need for assigning meaning to that language through the semantics.

This need arises an intention to provide a platform for rapid prototyping of new languages, allowing the rapid assignment of its semantics by transformation, in order to define a language in terms of another well defined and possibly under the specialist knowledge.

The current tools of model transformation, do not offer a balance between usability, validation and expressiveness.

In this work, we implement a tool based on graph grammars for definition and implementation of transformation rules. This involves designing a language for rules specification, construction of a visual editor and implementation of its engine.

Taking into account that this language and editor, wants to be a considerable assistance to the development of new languages, providing the assignment of semantics through model transformation, the usability will, necessarily, focus on simplicity and a smooth learning curve. Users may then develop languages without having to specialize in a transformation tool and thereby speeding up the process for defining language semantics.

Keywords: Model Transformation; Domain Specific Language; Software Language Engineering; Model Driven Development; Graph Transformation;

Conteúdo

Lista de Figuras

Lista de Tabelas

1

Introdução

Neste capítulo introduzimos o contexto do problema enquadrando-o com os objectivos do trabalho.

1.1 Contexto e Motivação

Actualmente, as Linguagens de Domínios Específicos (LDE) ganham protagonismo na Engenharia de Software devido à sua abordagem de resolução dos problemas com uso de linguagens ao nível de abstracção adequados e consequente possibilidade de soluções mais eficazes quanto à produtividade. Hoje em dia, o processo de engenharia de linguagens é suportado por várias ferramentas de metamodelação e transformação de modelos. No entanto, um dos aspectos que complica o desenvolvimento destas linguagens, é a complexidade associada à atribuição da sua semântica por tradução[15], sendo esse um dos principais pontos focados neste trabalho.

Com estes aspectos em mente e de modo a implementar os desenvolvimentos efectuados no âmbito do projecto *Building Adaptative Three-Dimensional Interfaces for Critical Complex Control Systems* (BATIC3S)[1][16] onde a abordagem passa por usar as transformações para definir a semântica das LDEs desenvolvidas no âmbito do projecto. Este facto permite uma mais rápida prototipagem das linguagens desenvolvidas para lidar com os problemas encontrados no contexto do projecto, uma vez que permite a reutilização de ferramentas de verificação e simulação nos formalismos considerados como alvo da tradução (por exemplo, Petri Nets), tornando-se uma parte fulcral do seu desenvolvimento. Assim, o engenheiro de linguagens de software não tem de se concentrar em esforço de definição das linguagens alvo, pois delega essa função

[1] BATIC3S - http://smv.unige.ch/research-projects/batic3s

para as comunidades dedicadas. É então necessário recorrer às ferramentas de transformação de modelos pois estas permitem a transformação entre duas linguagens através da definição de regras de transformação. No entanto, e embora existam várias alternativas, as ferramentas existentes muitas vezes ou não oferecem as funcionalidades pretendidas ou oferecem demasiadas funcionalidades sem aplicabilidade prática ou mesmo, são simplesmente demasiado complicadas para exprimir transformações simples necessitando de uma curva de aprendizagem muito grande. Deste modo pretendemos colmatar esta lacuna propondo uma ferramenta de transformação simples e eficaz para a execução rápida das transformações mais utilizadas. Desta maneira, obtemos mais disponibilidade para a definição das transformações em vez de perder demasiado tempo a tentar dominar uma outra ferramenta.

1.1.1 BATIC3S

BATIC3S é um projecto levado a cabo pelo grupo *Software Modeling and Verification Group* da Universidade de Genebra em parceria com a equipa do projecto *Software Languages Engineering for Requirements Specification and Design* (SOLAR) do Centro de Informática e Tecnologias de Informação (CITI)[2] da Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa. O projecto tem como objectivo a construção de interfaces adaptativas tri-dimensionais para sistemas complexos de controlo, onde se propõe uma metodologia de prototipagem rápida de sistemas a partir de especificações formais.

A metodologia do projecto BATIC3S assenta em primeiro lugar, na construção de linguagens para expressar modelos de interfaces adaptativas 3D para sistemas de controlo nos seus diferentes aspectos de modelação. Depois, tanto para a análise destes modelos como para a sua execução, foram criadas novas linguagens formais, que capturam apenas alguns aspectos sobre os modelos de interface originais (para fins de análise, execução, entre outros). Para que a metodologia possa ser utilizada na prática, as transformações entre linguagens têm um papel central em todo o processo, como se pode verificar pela figura 1.1. No topo desta figura (etiqueta 1, na figura) podemos verificar a existência de vários conjuntos de modelos que representam a especificação dos diferentes aspectos de modelação de um GUI de sistemas de controlo (casos de uso, perfis de utilizador, adaptabilidade, comportamento dinâmico, comportamento estático) . Sobre estes conjuntos de modelos são então desenhadas linguagens para solucionar conjuntos distintos de requisitos. Estas linguagens utilizam ferramentas de transformação para atribuição do seu significado (etiqueta 2, na figura) segundo redes de Petri (etiqueta 3, na figura). Tendo o modelo de redes de Petri devidamente criado, este gera um simulador que comunica com a GUI (etiqueta 4, na figura), sendo esta a fazer a ligação com o sistema real.

[2]CITI - http://citi.di.fct.unl.pt

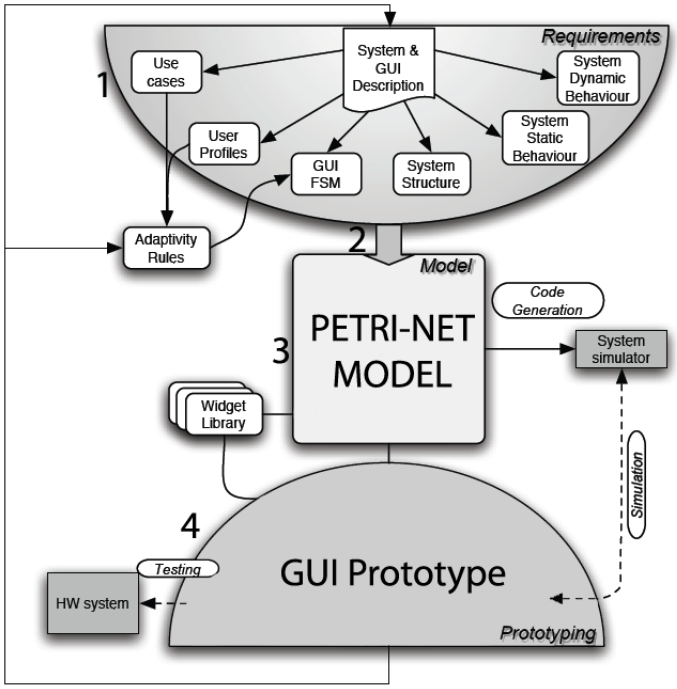

Figura 1.1: Metodologia BATIC3S baseada em [1]

1.2 Descrição do problema

Actualmente existe um variado número de linguagens de transformação que nos mostra uma gama variada de soluções bem sucedidas em diversos domínios. Em particular, de acordo com a metodologia Model Driven Development (MDD), o *Queries, Views and Transformations* (QVT)[17] surgiu como um primeiro standard para o apoio ao mapeamento semântico de linguagens especificadas através de metamodelos UML/MOF.

No entanto, para o caso particular de apoio à construção e desenvolvimento de LDEs baseadas em metamodelos, as linguagens/ferramentas existentes não são satisfatórias por diversos motivos:

- grande parte das linguagens de transformação existentes não apresentam elas próprias um metamodelo pelo que não é fácil raciocinar/analisar os modelos de transformação com vista a assegurar a correcção da transformação para todo e qualquer modelo de entrada pertencente a uma LDE, isto é, assegurar um modelo de saída válido.

- grande parte das linguagens de transformação existentes não apresentam um equilíbrio em termos de expressividade, usabilidade e validação[18].

- grande parte das linguagens de transformação existentes apresentam paradigmas de modelação pouco estruturados, e por vezes multi-paradigma (imperativo e declarativo) o que

dificulta não só o desenvolvimento do modelo de transformação (em particular para linguagens com um número considerável de regras), como qualquer perspectiva de análise sistemática (ou meta-nível) sobre os modelos de transformação criados como por exemplo a linguagem ATL[19].

1.3 Objectivos

Nesta dissertação, propomo-nos solucionar os problemas descritos anteriormente. Pretendemos assim desenhar uma linguagem de transformação, declarativa e baseada em gramáticas de grafos. A solução proposta tem como consequência a fácil atribuição da semântica a uma linguagem por transformações de modelos. No que respeita à implementação e devido ao contexto onde o trabalho se insere, o motor de transformação irá apenas lidar com modelos Ecore ([4]), de forma a ser compatível com outras ferramentas associadas à construção e desenho de LDEs, conformidade com tecnologias usadas actualmente de forma abrangente (nomeadamente *Eclipse Modeling Framework*) e também devido ao projecto onde se insere que desenvolve todas as suas linguagens utilizando esta plataforma.

A linguagem proposta terá um grande foco na usabilidade, porque a comunidade alvo é formada por especialistas de linguagens, que serão os responsáveis pela definição da semântica das LDEs. Pretende-se com isto fornecer uma plataforma simples de modo que esta seja utilizada como forma de especificação de semântica por transformação de modelos, ou seja, definir a semântica de uma nova linguagem através de transformações para uma outra, esta sim já especificada formalmente e com comportamento bem definido. Este factor de usabilidade estará intrinsecamente relacionado com o editor previsto para este motor de transformações. O editor será visual e diagramático para atribuição de uma sintaxe concreta à linguagem, de forma a tornar a solução usável (sendo devidamente demonstrado).

As funcionalidades do protótipo da ferramenta, não sendo o mais abrangente possível, apresenta as características essenciais de expressividade e usabilidade para a definição da semântica de uma LDE, dado ser esta a motivação que originou este trabalho.

1.4 Principais contribuições previstas

O trabalho aqui desenvolvido visa sobretudo a área de Desenvolvimento Orientado a Modelos, nomeadamente a Engenharia de Linguagens de Software. Será uma linguagem de transformações de modelos suficientemente completa e simples. Pretendemos que se torne no método mais simples de especificação de semântica de uma linguagem por transformação.

Prevê-se assim que esta dissertação contribua de forma considerável para o desenvolvimento de linguagens de software. Embora existam várias soluções disponíveis actualmente, tentamos colmatar algumas falhas encontradas, como por exemplo ao nível da usabilidade e correcção dos modelos transformados. Propomos novas funcionalidades úteis à definição de transformações

sem que com isto se comprometa a expressividade e simplicidade da ferramenta.

1.5 Estrutura do Documento

Este documento está organizado da seguinte forma:

Capitulo 2 Neste capítulo apresenta-se a área de Engenharia de Linguagens de Software, em particular conceitos que são necessários à compreensão do trabalho desenvolvido. Este trabalho assume maior importância nos conhecimentos teóricos, principalmente no que respeita às transformações.

Capitulo 3 É feito um estudo comparativo entre várias ferramentas de transformação já existentes, fazendo uma análise crítica em função dos objectivos identificados.

Capitulo 4 Este capítulo faz a apresentação formal da solução, nomeadamente através do seu modelo de características, metamodelo e definições de sintaxe concreta e respectiva semântica da linguagem.

Capitulo 5 Neste capítulo são apresentados os testes de avaliação à ferramenta feitos através de um teste experimental e de um caso de estudo de forma a efectuar uma apreciação global da ferramenta desenvolvida.

Capitulo 6 Finalmente faz-se uma apreciação do trabalho realizado e são sugeridos alguns caminhos para a evolução do mesmo.

2

Engenharia de Linguagens de Software

Neste capítulo pretende-se introduzir a área de Engenharia de Linguagens de Software, onde se insere o trabalho desta dissertação. Iremos também introduzir conceitos necessários para a compreensão e implementação da solução, nomeadamente no que respeita às transformações de modelos e de transformações de grafos, sendo estes, dois conceitos fundamentais no âmbito desta dissertação.

2.1 Desenvolvimento Orientado a Modelos

O desenvolvimento orientado a modelos (*Model Driven Development* - MDD)[20][21][2] consiste em pensar o desenvolvimento de uma forma mais abstracta de maneira a conseguir uma visão geral sobre a qual se torna mais simples raciocinar. No entanto, para melhor compreender esta abordagem, é necessário definir e compreender o que significa 'modelo' e 'metamodelo', e ainda do que se trata a *Unified Modeling Language* (UML) e *Meta Object Facility* (MOF) no contexto da Engenharia de Software.

MOF A *Meta Object Facility* [22] é uma especificação formal por parte da *Object Management Group* (OMG)[1], indicando um standard para a criação e manipulação de metamodelos. Esta definição fornece uma estrutura para linguagens de metamodelação.

[1] http://www.omg.org

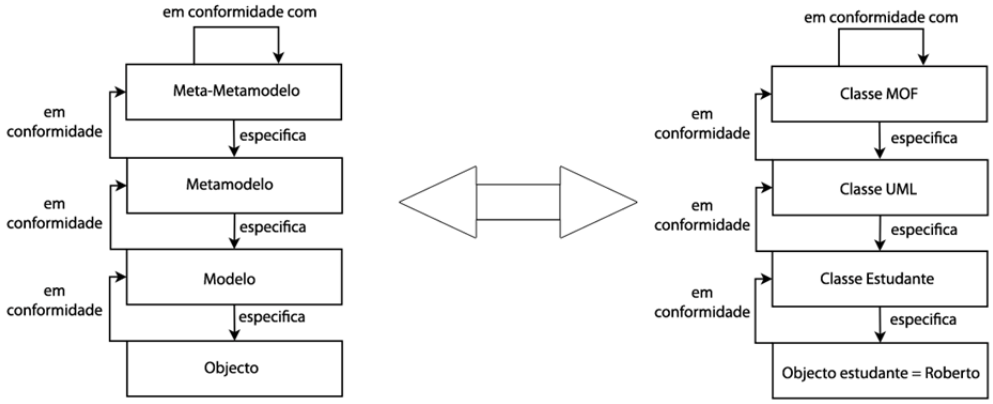

Figura 2.1: Quatro níveis de modelação, baseado em [2]

UML A *Unified Modeling Language* [23] é uma linguagem standard de metamodelação que permite a definição de novas linguagens de programação. Esta linguagem é composta por vários diagramas que permitem a descrição da estrutura de uma linguagem (diagramas de classes, diagramas de componentes, entre outros) e o seu comportamento (diagramas de actividades, de sequência, de casos de uso, entre outros).

Modelo Em Engenharia de Software, modelo significa um conjunto de diagramas formais que descrevem algo e são construídos como forma de análise de um determinado problema. Os modelos implicam uma abstracção a diversos níveis de detalhe sobre a implementação, normalmente feita através de diagramas UML sendo este um standard para a definição dos mesmos. O modelo é assim uma definição de mais alto nível que facilita a compreensão do problema por se encontrar mais próximo da compreensão humana, conseguindo uma visão mais simples potenciando uma melhor interpretação e validação do mesmo.

Metamodelo Um metamodelo é também um modelo, utilizado como forma de definir as regras de criação dos modelos de uma determinada linguagem. Permite assim definir instâncias bem formadas da linguagem modelada. A imagem 2.1 apresenta uma arquitectura dos quatro níveis de modelação definidos pela OMG. Temos o sistema real M0 (objecto estudante na figura 2.1), especificado por um modelo no nível superior M1 (Classe Estudante). Este modelo está em conformidade com o seu metamodelo no nível M2 (Classe UML) que por sua vez está em conformidade com um meta-metamodelo de nível M3 (Classe MOF), enquanto este estará em conformidade consigo próprio.

Esta abordagem ao desenvolvimento contribui de forma significativa para uma boa análise do problema em causa, no entanto o esforço necessário para a posterior implementação dificulta a adopção desta por uma comunidade dominada por programadores que se concentram no desenvolvimento directo para código. Os programadores duvidam da importância da

modelação da solução dado que esta muitas vezes torna-se desactualizada demasiado rápido, mesmo durante o processo de implementação da solução. De modo a contrariar esta situação, as ferramentas de desenvolvimento de software necessitam incluir automatismos de tarefas que permitam uma maior flexibilidade na implementação dos modelos, nomeadamente através de transformações de modelos.

As transformações de modelos poderão ser então o motor necessário para uma maior utilização da abordagem MDD, pois permite uma maior agilidade entre a modelação e a implementação de uma solução. Grande parte dos modelos podem assim ser transformados em código de forma automática facilitando a implementação e potenciando a evolução da solução, dado que uma alteração na modelação da mesma não implica necessariamente um esforço demasiado grande na programação, embora para que isto aconteça seja necessário que o processo de transformação seja automático e contenha métodos de verificação e validação.

2.2 Linguagens para Domínios Específicos

A Engenharia de Linguagens de Software assume uma importância crescente dentro da Engenharia de Software, principalmente pela maior relevância dada cada vez mais às Linguagens para Domínios Específicos[24][25][26].

Estas linguagens são menos abrangentes e concentram a sua expressividade no domínio do problema, permitindo uma maior produtividade do especialista do domínio. Permitem um nível de abstracção mais elevado, possibilitando expressar soluções ao nível do domínio, facilitando a aprendizagem e, consequentemente, permitindo que os especialistas do domínio possam validar ou até desenvolver programas numa determinada LDE. São linguagens que, habitualmente, aumentam a produtividade e fiabilidade nos resultados assim como a reutilização do conhecimento ao nível do domínio nomeadamente para a validação e optimização das soluções. As LDEs assentam ainda sobre as noções de modelos e metamodelos, descritas anteriormente, para assim descrever a respectiva sintaxe abstracta.

De forma a poder satisfazer os requisitos de forma sistemática, o engenheiro de linguagens efectua uma análise do domínimo identificando os termos e expressões mais relevantes para a linguagem em questão. Seguem-se as fazes de implementação e validação das linguagens, que fazem aumentar o custo de desenvolvimento, sendo um possivel problema para a criação de novas linguagens. Este processo pode ser minimizado através dos mecanismos MDD, nomeadamente utilizando transformações de modelos para agilizar implementações e reutilizar trabalho já realizado anteriormente.

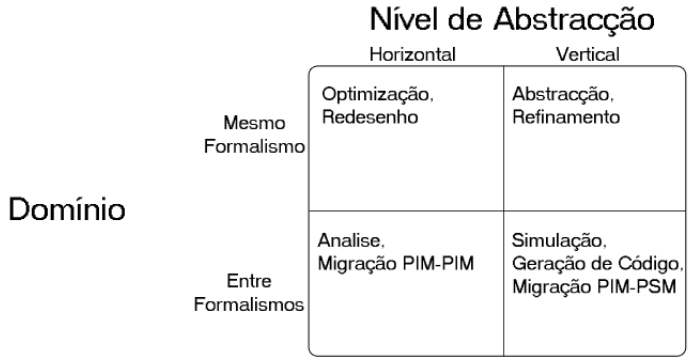

Figura 2.2: Representação das possíveis utilizações das transformações de modelos baseado em [3]

2.3 Transformações de Modelos

Actualmente os modelos desempenham um importante papel no desenvolvimento de software, em que standards como UML ou QVT sustentam a adopção mais generalizada da abordagem MDD. Com esta importância crescente dos modelos no desenvolvimento de software, surge como consequência as transformações de modelos[27].

No que toca ao desenvolvimento de linguagens, as transformações de modelos permitem aumentar ou diminuir o nível de abstracção da linguagem assim como possibilita uma mudança do próprio domínio sobre o qual a linguagem incide, por exemplo, através de migrações entre modelos independentes da plataforma (PIMs do inglês *Platform Independent Models*[28]), como pode ser visto no esquema da figura 2.2.

Este tipo de transformações é ainda importante pelo facto de facilitar a transformação rápida entre duas linguagens, nomeadamente, com o sentido de proporcionar uma prototipagem rápida de uma linguagem relativamente à atribuição da sua semântica, pois permite, através de regras de transformação de modelos, descrevê-la segundo termos de uma outra já formalmente definida, como podemos verificar, por exemplo, no projecto [16].

A figura 2.3 apresenta assim a ideia geral das transformações de modelos pretendidas, neste caso particular segundo o formalismo de transformações de grafos.

2.3.1 QVT - *Queries Views and Transformations*

O QVT é um standard, em MDD, para a transformação de modelos definida pela OMG. Este standard define a forma de transformar modelos em outros modelos que podem ser definidos através de metamodelos distintos. Este standard define que a definição da transformação é ela própria um modelo o que significa que estará também em conformidade com um metamodelo.

No entanto, o QVT apenas considera transformações entre modelos (do inglês *model-to-model*) o que deixa de fora todas as transformações de texto para modelos ou modelos para

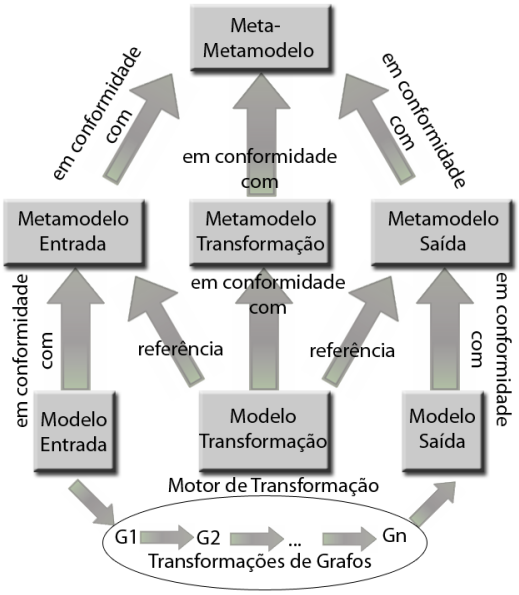

Figura 2.3: Esquematização das transformações de modelos através de transformações de grafos baseado em [3]

texto, embora estas sejam também importantes no contexto MDD.

2.3.2 Transformações de Grafos

Grafo Um grafo é composto por dois conceitos base, nós e vértices, onde os nós são estruturas que podem estar interligadas entre si através de vértices. Os grafos são uma forma eficiente de representação de determinados conceitos, pois permitem a representação de forma precisa, são programáveis e, acima de tudo, são visuais, o que facilita a compreensão de determinados sistemas.

Um metamodelo de uma linguagem pode ser representado como um grafo tal como todos os modelos criados a partir desse mesmo metamodelo, já que pode ser feita uma analogia entre nós, vértices (da parte dos grafos) e classes, associações (da parte dos metamodelos). Tendo os modelos representados por grafos torna-se mais fácil a definição de regras de transformação, seja para estender o próprio modelo ou transformá-lo num modelo de outra linguagem.

As gramáticas de grafos[29] providenciam um formalismo sobre o qual podem ser modeladas as transformações de grafos de uma forma matemática e precisa. Estas gramáticas fornecem vários tipos de transformações de grafos diferenciando-se entre si através da expressividade que possibilita em cada termo das suas regras. Definem várias metodologias de produção de grafos que possibilitam uma maior flexibilidade na adopção das características pretendidas à definição de grafos e consequente captura de padrões.

Qualquer transformação de grafos é efectuada através da aplicação de uma regra de reescrita

de grafos baseada, essencialmente, em quatro conceitos fundamentais:

- Pré-condição (*Left Hand Side Rule* - LHS),
- Pós-condição (*Right Hand Side Rule* - RHS),
- Condições de aplicação negativa (*Negative Application Condition* - NAC)
- Condições de aplicação positiva (*Positive Application Condition* - PAC).

A pré-condição representa um sub-grafo do grafo inicial que será utilizado para capturar os padrões que se pretende transformar, entrando aqui as pós-condições que serão o resultado que se pretende obter, por substituição ou criação das entidades presentes nesta última. As NACs, são condições que são expressas também através de um grafo que representa uma restrição aos casos pretendidos na transformação, isto é, mesmo existindo um padrão no grafo principal correspondente à pré-condição pretendida, a existência de uma NAC, implica que a transformação correspondente apenas seja executada caso não se verifique a condição expressa pela NAC[30]. Já as PACs obedecem às mesmas condições que as NACs no que diz respeito à sua sintaxe. No entanto, são condições que fazem sentido ser expressas na pós-condição, pois estas indicam que apenas se fará a transformação se existir (por oposição às NACs) no modelo de saída os elementos ou padrões representados por si.

As transformações de grafos por serem visuais proporcionam uma maior facilidade na descrição e percepção das regras, contudo possibilitam expressar de forma suficientemente completa todos os casos que pretendemos no âmbito desta ferramenta, embora a sua complexidade possa ser uma contrariedade à sua implementação, influenciando também a abordagem adoptada.

A figura 2.4 apresenta um exemplo de uma transformação de grafos, composto por LHS e RHS. A figura representa a acção de movimento do jogo Pacman. O padrão, que representa a posição actual do Pacman, é substituído por outro padrão, que representa a posição seguinte, o que neste caso em particular significa que o Pacman irá mover-se uma posição.

2.3.3 Abordagem de manipulação directa

Esta abordagem oferece uma representação de modelo interna e algumas APIs para manipulação deste. É habitualmente implementada como uma plataforma orientada a objectos, que pode fornecer uma infra-estrutura mínima para organizar as transformações (por exemplo, classe abstracta para transformações). No entanto, os utilizadores normalmente necessitam de implementar regras de transformação, ordenação, rastreabilidade, e outras características, muitas vezes desde raiz numa linguagem de programação de uso genérico (por exemplo, Java).[3]

2.3.4 Abordagem de transformação operacional

Este tipo de abordagem às transformações é semelhante à abordagem directa, no entanto, oferece um suporte mais dedicado às transformações de modelos. Uma solução tipicamente

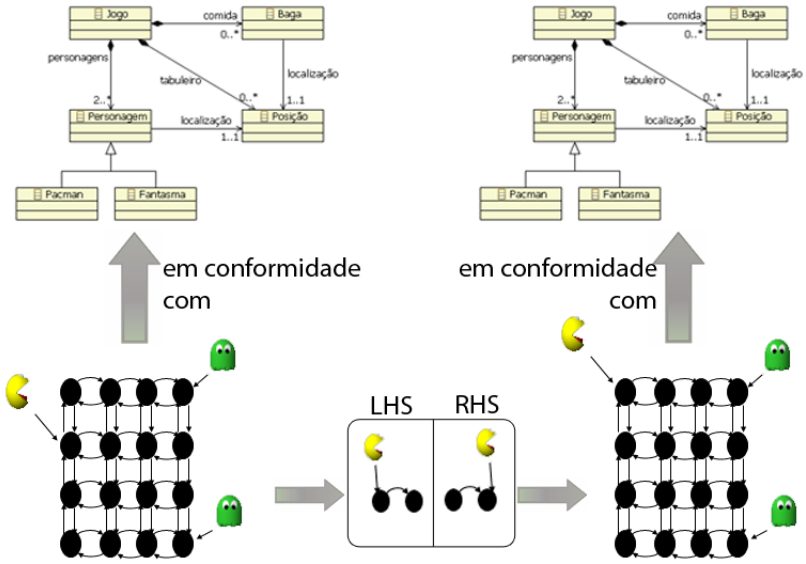

Figura 2.4: Exemplo de uma aplicação da reescrita de grafos

utilizada neste aspecto, é a extensão do formalismo de metamodelação utilizado com meios que permitem expressar computação. Um exemplo, seria estender uma linguagem de *queries* como a *Object Constraint Language* (OCL)[2], por exemplo, com conceitos imperativos. Soluções mais especializadas como a rastreabilidade podem ser oferecidas através de bibliotecas dedicadas.[3]

Um exemplo desta abordagem é mostrado mais adiante através da ferramenta de transformação Operational QVT.

2.3.5 Outras abordagens

Existem mais alternativas[31] à utilização de transformações de grafos. São estas por exemplo, *Common Warehouse Metamodel* (CWM), que providenciam conceitos *black-box* e *white-box* para definir transformações, definindo transformações através de construtores imperativos e declarativos descritos num ficheiro XML, ou a linguagem de programação *Mercury*, que é um linguagem puramente declarativa com características de captura de padrões.

No entanto devido à análise de vários artigos, em particular de comparação de ferramentas existentes [32][14], não foram consideradas como possibilidades para a implementação deste trabalho, pois são abordagens demasiado verbosas e consequentemente pouco legíveis, o que as impossibilita à partida de proporcionar uma boa usabilidade e produtividade no uso de transformações de modelos.

[2]http://www.omg.org/technology/documents/formal/ocl.htm

2.4 EMF - Eclipse Modeling Framework

O *Eclipse Modeling Framework* (EMF)[4] é uma plataforma de modelação para o ambiente de desenvolvimento Eclipse[3]. Caracteriza-se, em grande parte, por permitir o desenvolvimento de modelos e efectuar geração de código através destes. O EMF funciona então, como ponto intermédio entre as linguagens de programação e definições mais alto nível como seja o caso do UML.

Esta plataforma é assim extremamente útil para um desenvolvimento orientado a modelos pois permite uma junção entre a análise e implementação das aplicações desenvolvidas.

Um modelo EMF assenta sobre um subconjunto de diagramas de classes do UML, sendo utilizado para definição da sintaxe das linguagens desenvolvidas, e possibilita uma análise mais alto nível em relação ao código propriamente dito.

Ecore O Ecore é o modelo de implementação do EMF, servindo como forma de meta-modelação das linguagens desenvolvidas nesta plataforma. Este modelo é muito semelhante aos modelos UML, no entanto distingue-se por ser uma versão simplificada deste. O UML suporta modelação mais complexa do que o próprio EMF, como por exemplo o facto de permitir modelar o comportamento de uma aplicação através de diagramas de sequência ou actividade. O modelo Ecore representa uma versão mais simplificada do diagrama de classes UML em concordância com a definição do mesmo pelo EMF e está representado na figura 2.5. Com a utilização desta plataforma para a definição de modelos temos então uma transformação descrita na figura 2.6.

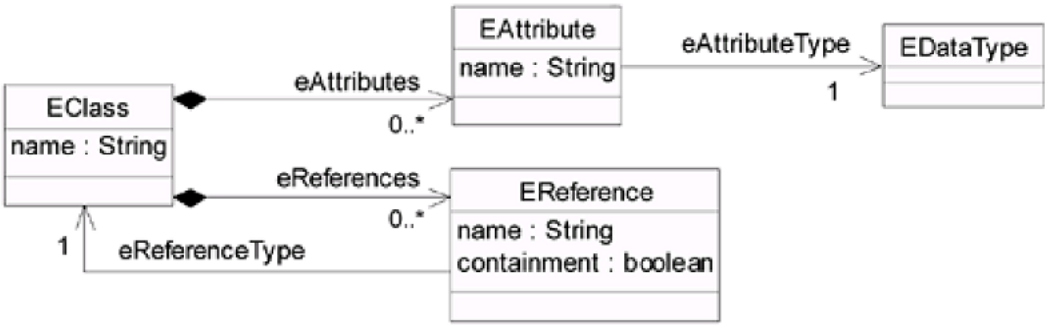

Figura 2.5: Conjunto simplificado de um modelo Ecore [4]

XMI Sendo o modelo Ecore o modelo de definição da sintaxe das linguagens, é necessário então uma forma de representar as instâncias desses metamodelos. Para esta tarefa, o EMF utiliza o formato XMI (*XML Metadata Interchange*), sendo este um standard para a representação

[3] http://www.eclipse.org

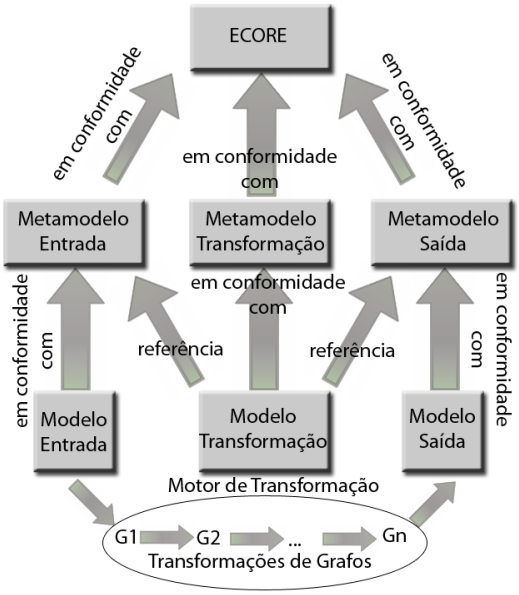

Figura 2.6: Esquematização das transformações de modelos através de transformações de grafos, com Ecore como modelo de Meta-metamodelação baseado em [3]

de modelos. Um ficheiro XMI gerado pelo Ecore é então uma serialização XML (*eXtensible Markup Language*) dos metadados utilizados pelo EMF.

2.5 GMF - Graphical Modeling Framework

A ferramenta *Graphical Modeling Framework* (GMF), é um *plugin* de extensão ao Eclipse que possibilita a implementação de editores visuais para modelos criados sobre a plataforma EMF. A imagem 2.7 representa, de uma forma simplificada, os passos de criação de um editor visual com esta ferramenta.

A escolha desta ferramenta tem por base a utilização de modelos EMF e a experiência prévia de utilização, assim como a existência de documentação e competência técnica dentro do próprio grupo SOLAR.

Será utilizada para a construção do editor visual de definição de regras de transformação de modo a possibilitar uma melhor usabilidade do próprio motor de transformações, sendo esse um dos pontos fundamentais deste trabalho.

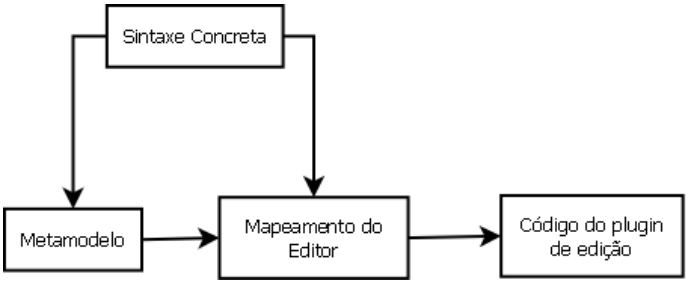

Figura 2.7: Simplificação da sequência de criação de um editor GMF

2.5.1 EuGENia

O projecto EuGENia[33] é uma ferramenta desenvolvida no âmbito do projecto Epsilon[4] de forma a possibilitar a criação de editores gráficos GMF com o menor esforço possível. Com isto possibilita uma rápida prototipagem de editores gráficos para linguagens desenvolvidas através do EMF, pois utiliza anotações feitas no próprio metamodelo para gerar de forma automática todo um editor e respectiva sintaxe concreta da linguagem desenvolvida. Funciona assim como uma abstracção sobre a plataforma GMF de modo a facilitar a passagem entre a sintaxe abstracta de uma linguagem e a respectiva sintaxe concreta. Embora esta ferramenta não implemente todas as potencialidades dos editores GMF, faz tudo o necessário para a implementação do nosso editor gráfico de forma bastante simplificada, requerendo um esforço mínimo para esta fase de implementação.

[4] http://www.eclipse.org/gmt/epsilon/

3

Análise de ferramentas de transformação

3.1 Introdução

Este capítulo apresenta uma comparação sobre as ferramentas de transformação existentes assim como uma breve análise sobre cada uma das ferramentas consideradas. Temos em especial atenção a usabilidade das ferramentas assim como algumas funcionalidades do modelo de transformação, como sejam os paradigmas utilizados e a definição das regras de transformação estando esta última também associada à sua usabilidade.

3.2 EMFTrans - Eclipse Modeling Framework Transformation Project

A ferramenta EMFTrans[34], apresenta uma interface gráfica e intuitiva para a descrição das regras de transformação. Utiliza a transformação por grafos como formalismo de definição de transformações, o que juntamente com a sua interface gráfica, visível na imagem 3.1, facilita a implementação das regras de transformação pretendidas. Permite a utilização de vários metamodelos assim como a criação (por transformação) de modelos compostos por instâncias de vários metamodelos. Do ponto de vista teórico, é utilizado a teoria algébrica das transformações de grafos como forma de determinar a ordem de aplicação das regras assim como a verificação de ciclos. Do ponto de vista de ferramenta de software, tem a vantagem de ser um *plugin* inserido na plataforma de desenvolvimento de software Eclipse, o que facilita a integração entre as variadas ferramentas de modelação e metamodelação existentes nesse suporte, nomeadamente o EMF, que é bastante utilizado nesta área como plataforma base para a criação de linguagens como visto na secção 2.4. No entanto, o EMFTrans recorre a um interpretador externo para

a interpretação das suas regras, e para que isto suceda, há a necessidade de gerar código Java para as regras pretendidas deixando depois ao cargo do especialista da linguagem a utilização do mesmo aquando da chamada do interpretador.

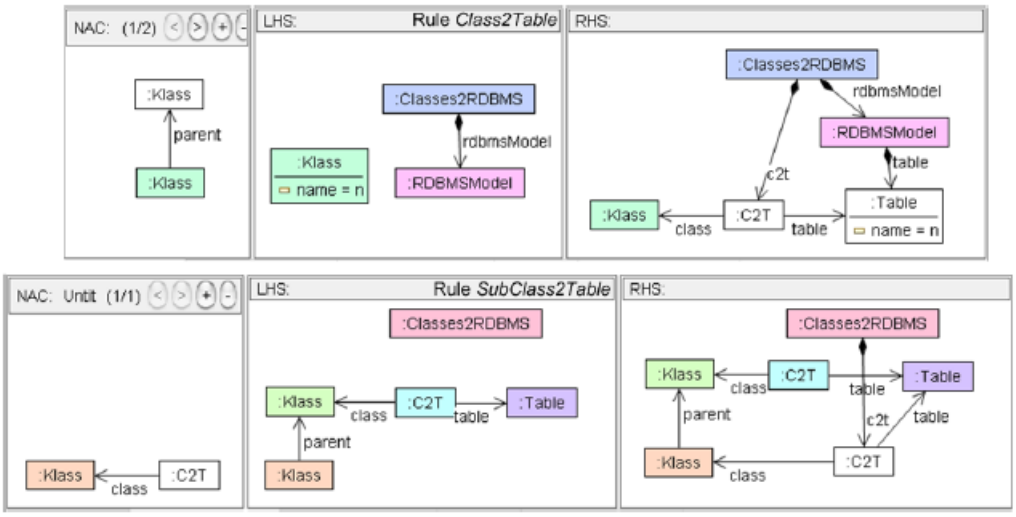

Figura 3.1: Exemplo de uma definição de regras em EMFTrans[5]

3.3 ATL - ATLAS Transformation Language

O ATL[19] é uma ferramenta de transformações de modelos com uma vasta utilização e com resultados satisfatórios por parte dos seus utilizadores mais experientes. Esta ferramenta apresenta uma definição textual de regras de transformação que assenta sobre um formalismo de transformações de grafos e, utilizado de forma correcta, poderá apresentar algumas mais valias a nível de expressividade da linguagem. É contudo, através de uma dependência excessiva da linguagem a regras OCL que a definição de regras se complica. A necessidade de utilização de código OCL em conjunto com a definição das regras, seja para navegação no modelo de transformação, ou como forma de apurar o resultado da transformação, torna de difícil leitura e validação as regras especificadas como solução de transformação, dificultando uma análise ou extensão futura da transformação. O OCL utilizado de forma excessiva, contribui ainda para uma dificuldade acrescida na análise sistemática e validação dos modelos de transformação.

A imagem 3.2[1] apresenta uma visão sobre a definição de regras em ATL, mostrando assim a dificuldade de análise das regras de transformação e da curva de aprendizagem necessária ao domínio da própria linguagem de transformação.

Em relação aos paradigmas utilizados, a linguagem ATL possibilita especificar regras no paradigma imperativo ou declarativo sendo assim uma tentativa de abranger um maior número

[1] http://www.eclipse.org/m2m/atl/usecases/SimplePDL2Tina/

de regras possíveis de implementar. No entanto, complica a tarefa ao utilizador pois a possibilidade multi-paradigma dificulta qualquer tentativa de análise do modelo de transformação criado. Esta situação torna ainda complicado o processo de aprendizagem pois requer ao utilizador o conhecimento de grande parte das técnicas disponíveis. Isto faz com que exista a necessidade de pensar nos múltiplos paradigmas ao invés de se concentrar no problema de semântica da linguagem.

```
module SimplePDL2PetriNet;
create OUT : PetriNet from  IN : pdl;
rule Process2PetriNet {
   from p : pdl!Process
   to pn : PetriNet!PetriNet (    nodes <- ,.. , arcs <- ,.. )
}
rule WorkDefinition2PetriNet {
   from wd : pdl!WorkDefinition
   to
      -- PLACES
      p_notStarted : PetriNet!Place (name <- wd.name + '_notStarted', nbJetons <- 1)
      ,...
      -- TRANSITONS
      , t_start : PetriNet!Transition ( name <- wd.name + '_start', temps_min <- 0, temps_max <- (0-1) )
      ...
      -- ARCS
      , a_nsed2s : PetriNet!Arc ( kind <- #normal, nbJetons <- 1, source <- p_notStarted, cible <- t_start )
      ...
}
rule WorkSequence2PetriNet {
   from ws : pdl!WorkSequence
   to
      a_ws : PetriNet!Arc ( kind <- #read_arc, nbJetons <- 1
            , source <- thisModule.resolveTemp(ws.predecessor,
               if ( (ws.linkType = #finishToStart) or (ws.linkType = #finishToFinish) )
                  then 'p_finished'
                  else 'p_started'
               endif )
            , cible <- thisModule.resolveTemp(ws.successor,
               if ( (ws.linkType = #finishToStart) or (ws.linkType = #startToStart) )
                  then 't_start'
                  else 't_finish'
               endif )
         )
}
```

Figura 3.2: Exemplo de uma definição de regras em ATL

3.4 GReAT - Graph Rewriting and Transformation

O GReAT[6] apresenta-se como uma ferramenta visual que implementa transformações de modelos através de transformações de grafos, definidas e exemplificadas na figura 3.3. Esta ferramenta utiliza como editor o GME (*Generic Modeling Environment*), também desenvolvido pela mesma equipa. Esta integração possibilita a utilização da mesma estrutura para o desenvolvimento desde os metamodelos, modelos e transformações. O GReAT utiliza também OCL, como forma de complementar a sua definição visual de regras, em alguns mapeamentos a efectuar entre modelos de origem e destino. Embora o GReAT tenha a vantagem de ser utilizado em ambiente totalmente visual, a execução da transformação é efectuada através do Visual Studio da Microsoft o que implica a sua dependência a uma ferramenta externa e proprietária. O Visual Studio é utilizado também para a geração de código C++ da transformação.

Também em relação à plataforma de desenvolvimento GME, surgem alguns problemas de utilização, devido ao facto desta não ser muito robusta. Os erros na ferramenta são frequentes

que podem surgir se, por exemplo, as regras de transformação não estiverem totalmente correctas e aquando da chamada do interpretador, o que pode acarretar perda de trabalho se este não for anteriormente guardado. O GReAT tem ainda outro problema que se deve ao facto do modelo transformado ser gerado num ficheiro XML de formato próprio, que impossibilita a utilização dos modelos por outras ferramentas.

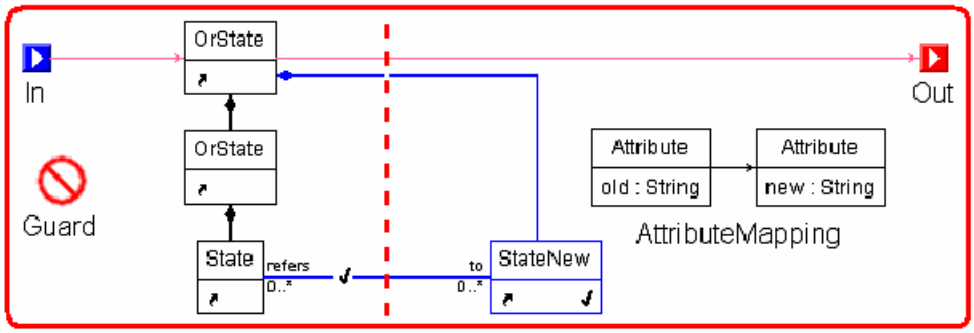

Figura 3.3: Exemplo da definição de uma regra em GReAT[6]

3.5 Viatra2

O Viatra2[7][35] é uma linguagem de transformação que utiliza transformações de grafos e máquinas de estados (*Abstract State Machines*) para a definição das regras, manipulação dos modelos e atribuição de ordem à aplicação das regras. A ferramenta que a implementa contém também uma componente de metamodelação de modo a definir os metamodelos dos respectivos modelos de entrada e saída. Esta componente de metamodelação visa sobretudo agregar vários conceitos de modelação como, por exemplo, o EMF, XML *Schemas* e Gramáticas EBNF.

Do ponto de vista do modelo de transformação, o Viatra2 suporta transformações unidireccionais e grande parte dos requisitos da especificação do QVT, que está próximo de se tornar um standard de transformações e, mesmo tendo uma linguagem de metamodelação própria, suporta através de *plugins*, a integração com metamodelos EMF, XML ou UML.

Em relação à usabilidade, esta é deixada para segundo plano, dada a quantidade de sub-linguagens existentes para as diversas funções, sendo todas elas executadas sobre a forma textual. Podemos ver um exemplo de definição de regras de transformação desta ferramenta na imagem 3.4.

3.6 Operational QVT

O *Operational QVT*[8] é uma ferramenta que tenta implementar a especificação QVT da OMG dado que esta especificação tem condições de se tornar um standard no que respeita às

```
gtrule liftAttrsR(in CP, in CS, in A) =
{
   precondition pattern cond(CP,CS,A,Attr) =
   {
      UML.Class(CP);
      UML.Class(CS);
      UML.Class.parent(Par,CS,CP);
      UML.Attribute(A);
      UML.Class.attrs(Attr,CS,A);
   }
   postcondition pattern rhs(CP,CS,A,Attr) =
   {
      UML.Class(CP);
      UML.Class(CS);
      UML.Class.parent(Par,CS,CP);
      UML.Attribute(A);
      UML.Class.attrs(Attr2,CP,A);
   }
}
```

Figura 3.4: Exemplo da definição de uma regra em Viatra2[7]

transformações.

A ferramenta tem uma interface textual, que não facilita a usabilidade, e está pensada sobretudo para transformações de modelos complexos. Utiliza um paradigma imperativo colmatado com algumas regras OCL, inclusive com alguma extensões, para uma melhor navegabilidade do modelo de transformação.

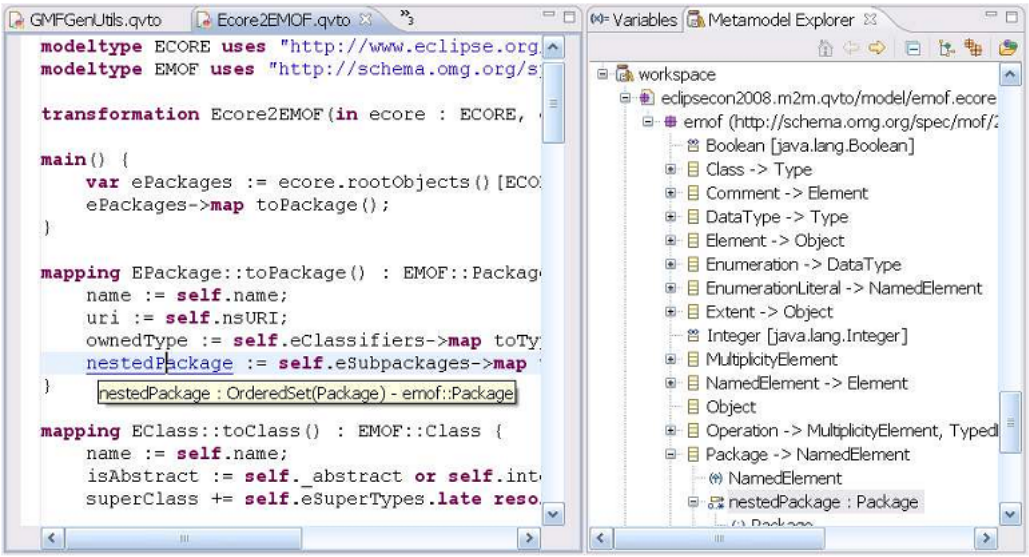

Figura 3.5: Exemplo da interface da ferramenta Operational QVT[8]

3.7 PROGRES

O PROGRES[9], tem bastante tempo de desenvolvimento tanto da linguagem, como da própria ferramenta. Embora haja o aparecimento de muitas outras ferramentas actualmente, o PROGRES continua a ser suportado e é assim um dos pioneiros das ferramentas de transformação de modelos.

A ferramenta utiliza definições de modelação multi-paradigma, com gramáticas de grafos como base de especificação da sua linguagem. O PROGRES apresenta assim uma interface visual, com complementos textuais, de definição de regras conjugada com um motor de aplicação de padrões, de forma não determinística, assim como funcionalidades de *backtracking*. A interface de definição de regras de transformação desta ferramenta encontra-se apresentada na figura 3.6.

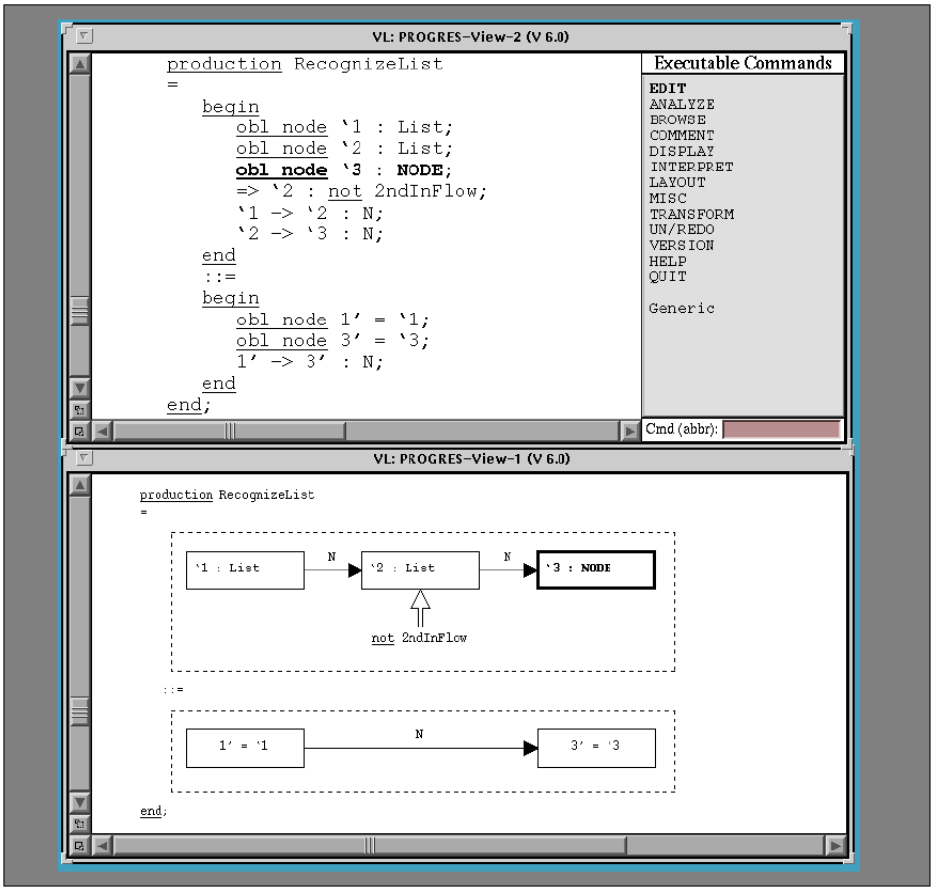

Figura 3.6: Exemplo da definição de regras no PROGRES[9]

3.8 Fujaba

A ferramenta Fujaba[36][37], é uma ferramenta com uma vasta comunidade de utilizadores. Como muitas outras propostas, o Fujaba pretende ser uma ferramenta integrada de desenvolvimento, nomeadamente com uma relação próxima entre UML e Java, com mecanismos de geração de código através dos diagramas UML definidos.

Em relação às transformações, o Fujaba específica as suas transformações como transformações de grafos, em particular *triple graph grammars* de modo a assegurar transformações bidireccionais de modelos, como ilustrado pela figura 3.7. A especificação gráfica é executada de forma incremental de modo a permitir a propagação de alterações entre modelos relacionados. Permite assim uma especificação visual das regras de transformação sobre um paradigma declarativo. Tem ainda mecanismos de rastreabilidade associados às *triple graph grammars*, necessárias para a manutenção da consistência dos modelos envolvidos nas transformações.

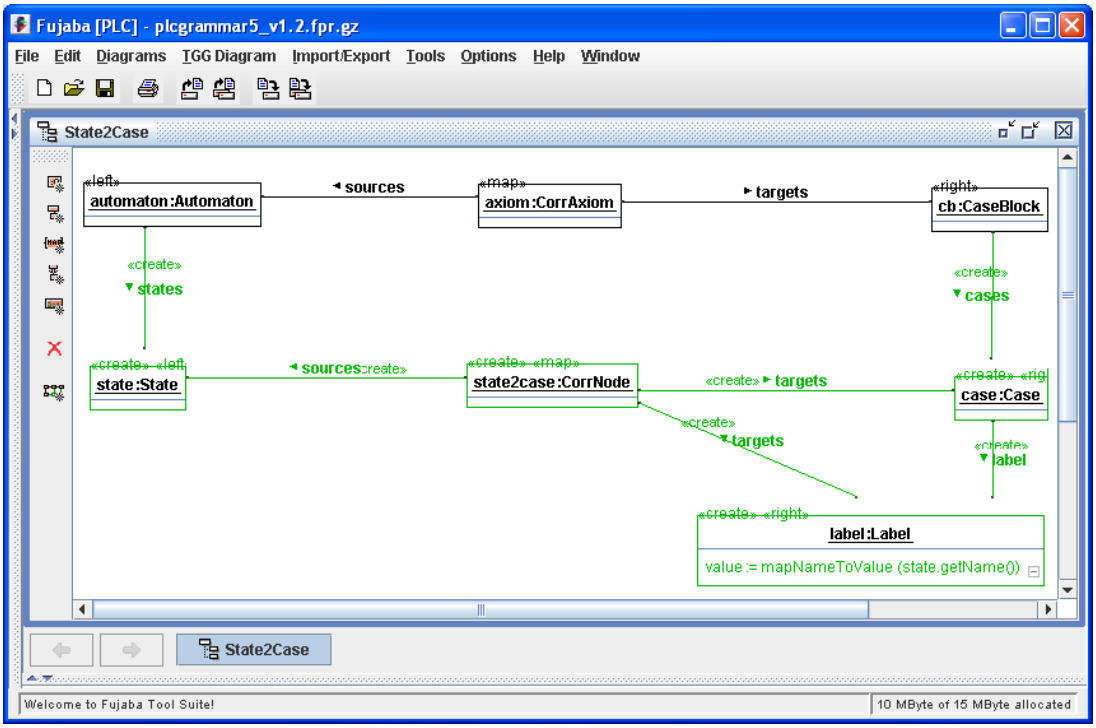

Figura 3.7: Exemplo da definição de regras em Fujaba [10]

3.9 Moflon

A ferramenta Moflon[38], pretende apresentar uma solução integrada de metamodelação, modelação e transformações, respeitante aos standards da OMG. No que respeita às transformações, esta ferramenta utiliza o motor de transformação do Fujaba baseado em transformações de grafos. A definição de regras é feita de modo visual, através de gramáticas de grafos, sendo complementada com restrições OCL, e pode ser visto um exemplo da sua utilização através da figura 3.8.

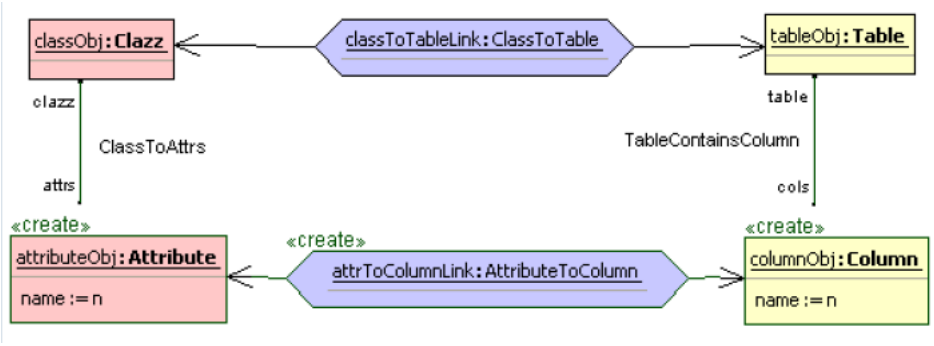

Figura 3.8: Exemplo da definição de regras no Moflon[11]

3.10 IBM: Rational Software Architect

A ferramenta Rational Software Architect[2], é uma proposta da IBM que congrega um grande conjunto de soluções para o desenvolvimento de software, em particular para desenvolvimento orientado a modelos, nas quais está inserida uma ferramenta de transformação de modelos.

Esta solução da IBM tem uma abordagem interessante ao nível da definição das regras de mapeamento das transformações, nomeadamente de atributos, o que favorece substancialmente a usabilidade da mesma. Este mapeamento entre entidades dos diferentes modelos é feita de forma visual, facilitando assim a sua utilização. No entanto, falta um método de captura de padrões nos modelos de entrada de modo a dar uma maior abrangência às transformações entre modelos. As transformações permitidas assentam sobretudo sobre mapeamentos entre entidades de diferentes metamodelos, permitindo contudo um refinamento das transformações através de código Java.

[2]http://www-01.ibm.com/software/awdtools/swarchitect/websphere/

3.11 Kermeta

Esta ferramenta integrada na plataforma de desenvolvimento Eclipse, tem como objectivo oferecer uma nova linguagem de metamodelação e assim proporcionar o desenvolvimento de linguagens pretendendo assim capturar de forma concisa todos os aspectos de uma linguagem de modelação, incluindo a sua sintaxe e semântica.

O Kermeta apresenta uma linguagem de sintaxe imperativa, orientada a objectos, com tipos estáticos e em conformidade com o MOF. Deste modo, a ferramenta posiciona-se de modo a capturar os pontos de convergência dos diferentes conceitos como pode ser observado pela figura 3.9.

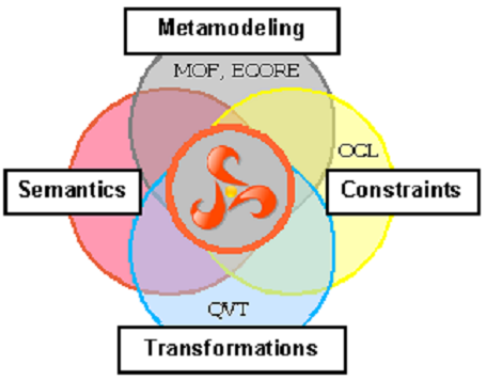

Figura 3.9: Posição do Kermeta na modelação [12]

O Kermeta utiliza ainda esta linguagem para a definição das próprias regras de transformação, oferecendo ao utilizador uma forma imperativa de definição de padrões como é visível pela figura 3.10.

3.12 Conclusão

Partindo então da categorização de características baseada em [14] e [32], comparamos as ferramentas analisadas nos pontos que consideramos mais importantes a nível de usabilidade, correcção e expressividade. Esta comparação permite-nos extrair informação necessária e importante para a implementação de uma nova linguagem e perceber quais os pontos que podem, efectivamente, ser melhorados, indo assim de encontro ao objectivo que se pretende para a solução que propomos. A tabela 3.1 é então construída através dos artigos consultados para a análise feita anteriormente para cada uma das ferramentas assim como pelos testes feitos em algumas delas, tendo sempre em conta os objectivos definidos inicialmente nesta dissertação.

Algumas das ferramentas analisadas, têm como objectivo abranger o maior número de funcionalidades possível, dando assim um maior liberdade ao utilizador e garantindo muitas vezes

```
createColumnFromAttribute(theAttribute: source_model::Attribute;
                          theTable : target_model::Table;
                          namePrefix : Standard::String) :
target_model::Column
{
   theColumn  : target_model::Column;
   name       : Standard::String;

   name := namePrefix.concat(

            theAttribute.name.oclAsType(!Standard::String!));
   if
(theAttribute.type.oclIsKindOf(!source_model::PrimitiveDataType!))
   {
      theColumn := new target_model::Column();
      theColumn.name := name;
      theColumn.type := theAttribute.type.name;

      associate (cols := theColumn : target_model::Column,
                 owner := theTable : target_model::Table );
      if theAttribute.is_primary
      {     // we also need to associate it as a pkey
         associate ( pkey := theColumn : target_model::Column,
                             pkeyreferers:= theTable :
target_model::Table );
      }
   }
   return theColumn;
}
```

Figura 3.10: Exemplo de uma definição de regra de transformação em Kermeta[13]

Ferramentas	Editor		Paradigma		Direccionalidade		Expressividade		Interpretador		Compatível
	Textual	Visual	Declarativo	Imperativo	Unidireccional	Bidireccional	Camadas	Navegabilidade	Externo	Interno	EMF
EMFTrans		✓	✓		✓		✓	RRR	✓		✓
Great		✓	✓		✓			OCL	✓		
ATL	✓		✓	✓	✓			RRR e OCL		✓	✓
Viatra2	✓		✓	✓		✓		RRR		✓	✓
oQVT	✓			✓		✓		OCL	✓		✓
PROGRES	✓	✓	✓	✓	✓			OCL	✓		
Fujaba		✓	✓			✓		RRR e OCL	✓		
Moflon		✓	✓			✓		RRR e OCL	✓		
IBM Rational		✓	✓		✓		✓	Java		✓	
Kermeta	✓			✓			✓			✓	✓
DSLTrans		✓	✓		✓		✓	AI e RR		✓	✓

Tabela 3.1: Tabela comparativa de características das várias ferramentas utilizadas com base em [14].

um maior nível de expressividade. A nossa proposta no entanto pretende conter um conjunto mais restrito de funcionalidades, mas bem definido em função dos objectivos de usabilidade e do contexto onde nos inserimos. É particularmente importante manter o número de escolhas do utilizador reduzido, de forma a proporcionar apenas uma única forma de definir uma determinada transformação, evitando assim ambiguidades e dificuldades de verificação do modelo de transformação detectadas, por exemplo, num trabalho anterior realizado no contexto do projecto BATIC3S[39].

Na tabela 3.1 temos várias características de comparação, todas elas relacionadas com os objectivos traçados inicialmente nesta dissertalçao. Em relação ao objectivo da usabilidade, utilizamos como método comparativo, o editor de cada ferramenta, nomeadamente se utilização editores visuais ou textuais, pois achamos ser um factor que influência a facilidade de utilização da mesma. Ainda no aspecto da usabilidade, temos o interpretador utilizado pela ferramenta.

Muitas vezes estas utilizam interpretadores externos, o que aumenta o número de tarefas a executar para processar efectivamente a transformação e desta forma prejudicam, não só a sua usabilidade como a própria produtividade do utilizador.

Temos também como ponto essencial das ferramentas de transformação, a sua expressividade, na tabela representada pela capacidade de compartimentar as regras por camadas e a navegabilidade. As camadas possibilitam ao utilizador uma forma de ter algum controlo sobre a sequência de transformação. A navegabilidade consiste na capacidade de construção de modelos complexos, através de composição de modelos anteriormente transformados. Nesta característica, o facto de muitas ferramentas optarem por utilizar restrições OCL sobre as suas regras, afecta um outro factor como a análise do modelo de transformação. Esta análise está sobretudo dependente dos paradigmas utilizados, pois se a ferramenta possibilitar a definição de regras em paradigmas distintos e juntamente com a utilização de regras OCL, impossibilita qualquer tipo de raciocínio sobre o modelo de transformação definido, nomeadamente por terceiros.

Devido ao contexto onde nos inserimos é particularmente importante que a ferramenta seja também compatível com modelos EMF, dado existir um vasto trabalho já realizado nesta plataforma.

Algumas das ferramentas estudadas implementam linguagens com características semelhantes às que pretendemos para a linguagem desenvolvida nesta dissertação. No entanto a própria ferramenta, muitas vezes dificulta um maior aproveitamento das reais capacidades da linguagem de transformação definida, nomeadamente no que respeita à interacção com o utilizador tendo consequências ao nível da sua usabilidade. Um exemplo desta situação é por exemplo o MOFLON, que tendo uma linguagem suficientemente bem estruturada e com capacidade de produzir transformações bidireccionais, peca pela sua ferramenta que falha ao nível da interacção do utilizador e por alguma instabilidade que leva a perda de trabalho.

Devido a condicionantes de espaço ocupado pela tabela abreviamos alguns dos conceitos, indicando em seguida o seu significado:

RRR Referência para Resultados de Regras - Indicação explicita sobre o resultados de uma determinada regra para controlo da construção de modelos mais complexos.

AI Associações Indirectas - Forma abstracta de capturar caminhos nos padrões das regras, utilizada na nossa ferramenta e explicada em mais detalhe no capítulo seguinte.

RR Restrições Retroactivas - Forma de capturar elementos transformados anteriormente, utilizada na nossa ferramenta e explicada em mais detalhe no capítulo seguinte.

4

DSLTranslator

4.1 Modelo de Características

Um modelo de características (do inglês *Feature Model*[40]) apresenta de forma diagramática as funcionalidades pretendidas numa determinada linguagem/aplicação de software. Não sendo demasiado específica, esta representação apresenta uma abstracção sobre a própria modelação da solução e permite uma visualização mais geral de todas as características e possíveis relações ou restrições entre elas.

Deste modo apresentamos o nosso Modelo de Características, como forma de introduzir a proposta de solução e oferecer uma visão geral das funcionalidades pretendidas.

A solução proposta não será mais completa a nível de quantidade de funcionalidades, pois como visto no capítulo 3, existem várias soluções com muitas funcionalidades e permitem uma grande expressividade aos seus utilizadores. No entanto, pretendemos limitar esse tipo de escolha para que, seja oferecida aos utilizadores uma ferramenta usável contendo as funcionalidades mais utilizadas de modo a que estes possam implementar a sua linguagem de forma simples e eficaz.

Unidirecional O motor permitirá transformações unidireccionais, significando isto que será capaz, através do mesmo conjunto de regras definidas, de transformar A $\rightarrow$ B mas não B $\rightarrow$ A, sendo A e B duas instâncias distintas de linguagens diferentes ou não.

Editor Gráfico A ferramenta terá um editor gráfico, sendo esta a interface disponibilizada ao utilizador para a definição das suas regras de transformação, com uma sintaxe concreta

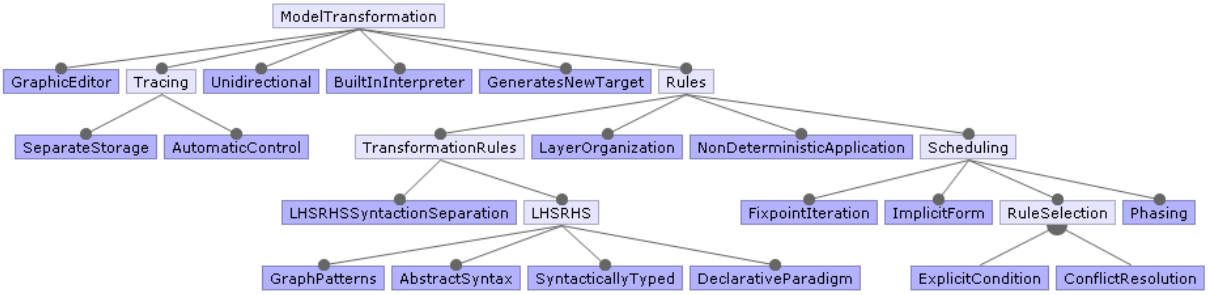

Figura 4.1: Modelo de Características da linguagem de transformação DSLTranslator

semelhante à transformação de grafos.

Rastreabilidade A rastreabilidade será um procedimento interno ao motor de transformações de modo a guardar um histórico da criação das respectivas entidades e associações do modelo de saída de modo a permitir um possível raciocínio sobre as mesmas em função da sua origem.

Interpretador O interpretador das regras de transformação será interno à ferramenta não necessitando assim de utilizar ferramentas externas para o processamento efectivo da transformação.

Modelo Saída Os modelos de saída resultantes das transformações serão modelos criados de raiz pelo motor de transformação. Esta opção, pretende sobretudo evitar perda de informação no modelo de entrada.

Regras As regras estão decompostas em várias componentes. São assim regras expressas sob a forma de grafos, com separação sintáctica entre LHS e RHS (metamodelos distintos), com sintaxe abstracta e definidas de forma declarativa. As regras estão organizadas em camadas (*layers*) de modo a conferir uma ordem temporal à sua execução. A aplicação de cada regra em cada camada é realizada de forma não determinística segundo os número de soluções presentes no modelo de entrada, ou seja, para cada solução existente no modelo de entrada é criado o padrão correspondente no modelo de saída. Como o modelo de entrada é estático durante todo o processo de transformação e os padrões de captura são aplicados sobre este, existe garantia de término da transformação.

4.2 Metamodelo

Embora pretendamos uma linguagem de transformação suficientemente completa para que possa ser utilizada de forma significativa pelos construtores de novas linguagens, é também nossa pretensão mantê-la o mais simples possível de modo a facilitar e simplificar o raciocínio sobre a mesma, em particular sobre o seu metamodelo apresentado na figura 4.2.

O metamodelo demonstra a organização pretendida da linguagem de transformação. Temos então o modelo de transformação (*Transformation Model*) que contém várias camadas de regras. Essas camadas podem então ser sequenciais ou de refinamento, em que se descreve as diferenças mais adiante neste relatório, e cada camada terá uma referência ao metamodelo correspondente, sendo que estes serão herdados de camadas anteriores.

Cada camada tem um conjunto de regras, e cada regra terá dois tipos de modelos:

- um modelo de entrada (LHS em gramáticas de grafos)
- um modelo de saída (RHS em gramáticas de grafos)

A principal diferença nestes dois tipos de modelos, será a possibilidade de conformidade com metamodelos distintos, embora não necessariamente, pois ambos são constituídos por classes, associações e atributos.

Existe ainda uma outra entidade denominada *AttributeMap*, que pretende definir os mapeamentos de atributos das várias classes segundo um conjunto limitado de operações a efectuar sobre os atributos do modelo de entrada.

4.3 Modelos

Os modelos utilizados estarão definidos em formato XMI que fornece uma forma standard de especificação dos mesmos, nomeadamente, através da plataforma EMF utilizada, e em que se baseia esta ferramenta. A escolha destes modelos EMF assenta sobretudo sobre o princípio de utilização de standards, pois o EMF implementa a especificação MOF (*Meta-Object Facility*) desenvolvida pelo OMG (*The Object Management Group*) como forma de definir metamodelos, gerar XML *schemas* para permuta ou gerar interfaces de manipulação de modelos[20]. Os modelos de origem terão de ser necessariamente válidos segundo o seu metamodelo definido em EMF, pois caso isso não se verifique, a transformação não será processada por erros de sintaxe. A ferramenta gera novos modelos de saída consistentes com os metamodelos de saída especificados, não efectuando alterações ou actualizações ao modelo de entrada através das transformações definidas pelas regras. Ainda que o metamodelo de saída seja o mesmo que o de entrada, o resultado será sempre guardado num novo modelo.

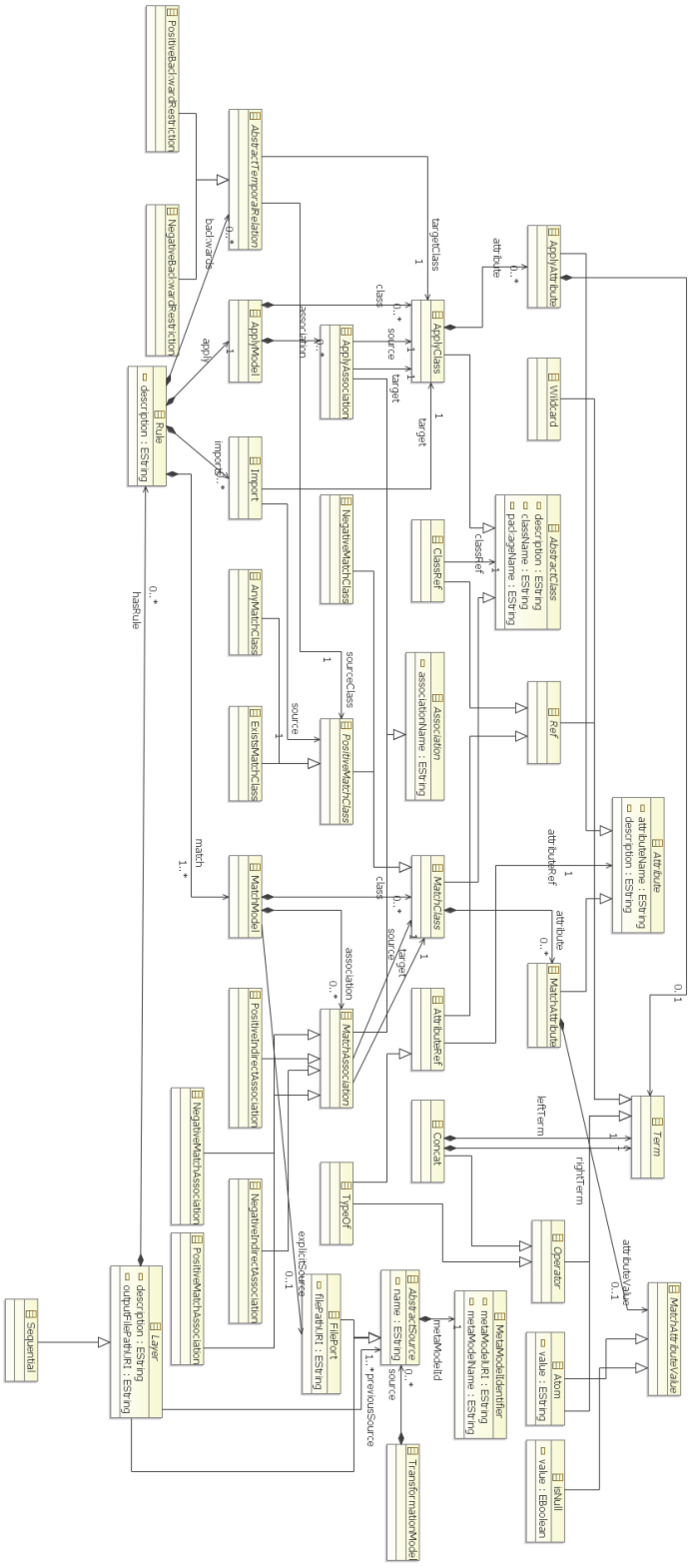

Figura 4.2: Metamodelo da linguagem de transformação DSLTranslator

4.4 Regras de transformação da linguagem

A linguagem proposta, baseia-se essencialmente, na aplicação de regras de transformações de grafos para a definição formal das transformações a efectuar sobre um determinado modelo. As transformações de grafos permitem-nos assim abranger um grande número de transformações possíveis, assim como fornecer uma forma simples de definição das mesmas. Isto garante-nos então um bom nível de expressividade ao mesmo tempo que nos fornece todo um formalismo robusto de apoio à implementação e definição das próprias transformações.

As regras de transformação serão definidas pelo utilizador através de um editor visual e sem a necessidade de escrever código de qualquer espécie (Java, OCL, etc), facilitando assim a sua tarefa. No entanto, embora as regras sejam definidas utilizando a teoria de grafos num editor gráfico, estas serão processadas através de operadores relacionais, pois estes fornecem uma maior simplicidade às operações necessárias ao mesmo tempo que garantem captura dos padrões nos modelos de origem.

4.4.1 Organização das Regras

A aplicação das regras sobre o modelo de origem é feita de forma não determinística. Desta forma achamos necessário a criação de camadas, de forma a possibilitar ao utilizador algum controlo sobre a ordem de aplicação de cada conjunto de regras.

4.4.1.1 Camadas

Uma camada de regras é um conjunto de regras. As regras dentro de cada camada continuam a ter uma aplicação não determinística, no entanto cada camada tem um antecessor que poderá ser uma outra camada de regras ou um modelo, nomeadamente através de um ficheiro XMI.

As camadas surgem pela necessidade de atribuição de alguma ordem à aplicação das regras de modo a possibilitar um maior controlo à aplicação das mesmas, pois poderá ser necessário aplicar um conjunto de regras em primeiro lugar para posteriormente ser possível compor o modelo de saída, sendo este construído de forma incremental.

Este conjunto de regras aplica os padrões de captura (*Match*) ao modelo de entrada original e produz resultados incrementais aos modelos de saída das regras anteriores como pode ser observado na através da figura 4.3.

4.5 Navegabilidade do Modelo de Transformação

As regras, sendo regras de transformação de grafos, estão divididas, necessariamente, em três partes distintas. A LHS, que define os padrões a capturar, a RHS, que indica a transformação pretendida, e as NACs que indicam restrições aos padrões LHS que pretendemos capturar.

A nossa proposta, ainda que contenha todos estes elementos, pretendemos complementar com alguns conceitos novos, que julgamos bastante úteis na construção da semântica de uma

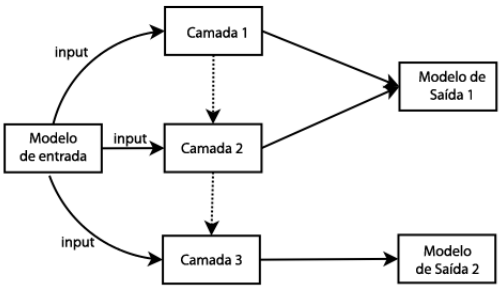

Figura 4.3: Esquema representativo da organização de camadas

nova linguagem através de transformações de modelos.

Em primeiro lugar, efectuamos uma pequena alteração à forma como as NACs serão apresentadas dado que iremos agregar as noções de LHS com as NACs, pois as NACs são restrições às LHS, ficando assim mais intuitiva a sua utilização como se pode ver pela figura 4.4 em que a ligação A $\rightarrow$ C representa uma NAC à regra A $\rightarrow$ B.

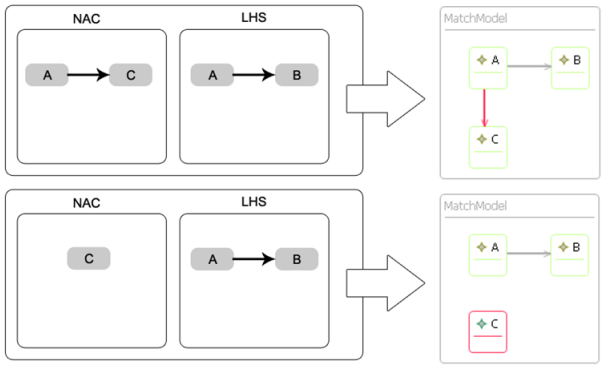

Figura 4.4: Representação da LHS com NACs

As soluções apresentadas em seguida permitem a construção de modelos complexos pois funcionam como mecanismo de restrição de padrões. Dadas as abstracções que fornecem sobre o modelo de transformação, julgamos essenciais para o sucesso da nossa ferramenta de transformação.

4.5.1 Associação Indirecta

Juntamente com as NACs, pretendemos adicionar outra forma de restrição aos padrões que será uma restrição por associação indirecta de duas entidades. Esta restrição funciona, assim como as NACs, como restrição à LHS significando que esta apenas considerará os padrões que contenham uma ligação entre as duas entidades, mesmo que pelo meio estejam muitas outras e que se encontra representada pela figura 4.5. No entanto, de forma a podermos garantir que

existe apenas um único caminho entre estas duas entidades e assim evitar possíveis complicações devido a ciclos que poderiam existir, são apenas consideradas ligações de agregação entre as entidades presentes nesse caminho. Assim podemos garantir que não existem entidades auto-contidas evitando a existência de ciclos durante todo o caminho.

Para esta funcionalidade em particular, prevê-se uma complexidade acrescida para a computação da mesma, dado que, para modelos muito grandes, a geração de todas as suas dependências poderá tornar-se uma tarefa computacionalmente pesada. No entanto, esta tarefa poderá ser minimizada fazendo a geração a pedido, pois é razoável considerar que este conceito não será usado de forma massiva, embora a cada geração da sub-árvore de dependências geradas, essa geração é armazenada para um possível uso futuro evitando uma nova computação das dependências.

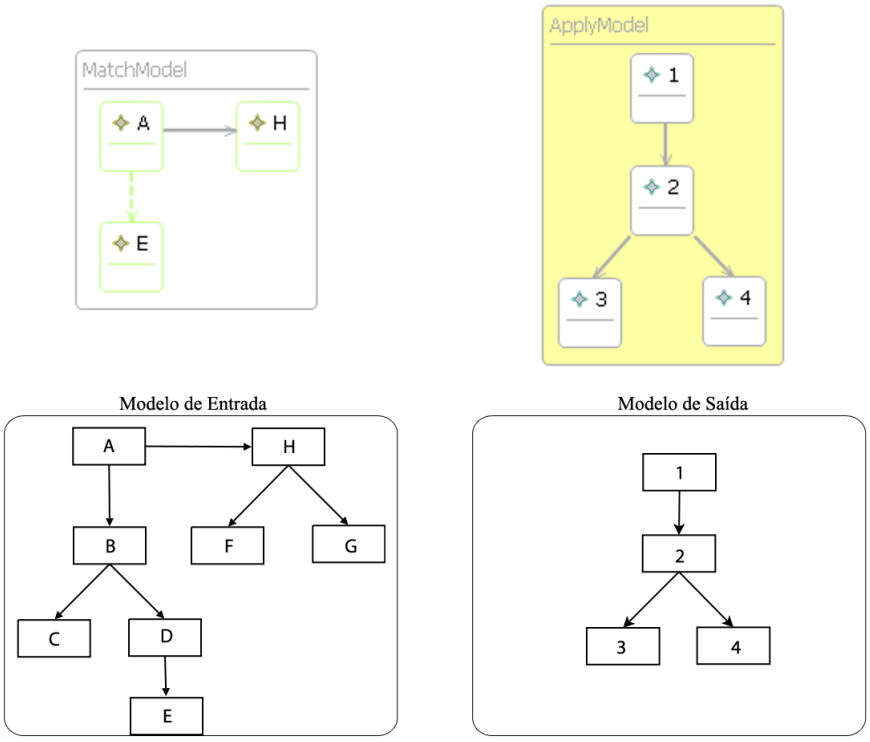

Figura 4.5: Associação indirecta na LHS

4.5.2 Restrição Retroactiva

No que diz respeito às suas restrições, é ainda adicionado o conceito de histórico que representa a origem de determinada entidade, permitindo assim um maior controlo e navegação nas regras. Este conceito de histórico será incluído nas definições das várias componentes da regra

(LHS, RHS e NAC) interligando as mesmas como demonstrado pela figura 4.6. Isto significa que uma entidade é transformada a partir de uma ou mais entidades de origem. Este histórico será processado de forma implícita pelo motor de transformação a cada passo de aplicação das regras, pois a linguagem de transformação é capaz de determinar as entidades de origem e assim guardar estas como sendo a procedência das entidades geradas.

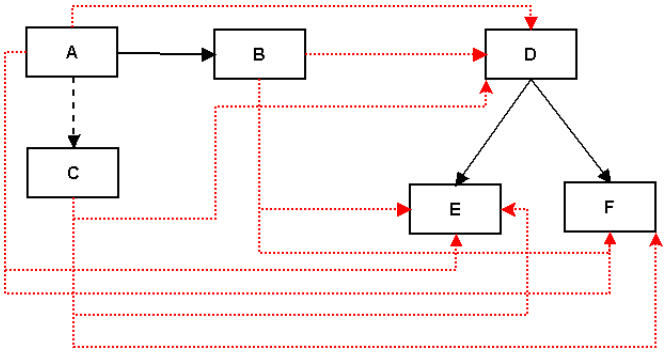

Figura 4.6: Histórico das entidades

Associado ao conceito de histórico temos a possibilidade de limitar de novo a aplicação das regras, pois o utilizador pode expressar a intenção de que pretende aplicar um padrão RHS de uma determinada regra, mas apenas se uma ou mais entidades destino estão já criadas e têm como origem uma ou mais entidades presentes no corpo da regra. A sintaxe concreta desta funcionalidade encontra-se demonstrada pela figura 4.7.

É também possível restringir o número de soluções através restrições impostas sobre os atributos das classes já existentes. Desta forma, é possível ser mais específico em relação às classes que pretendemos capturar através da condição de retroactividade. Estas condições são definidas como um atributo normal, mas se a classe estiver uma condição de retroactividade esse atributo funciona como restrição. Não é possível criar novos atributos a classes já existentes.

4.5.3 Importação

Pelo contexto onde esta ferramenta é desenvolvida (semântica por transformação), existe muitas vezes a necessidade de importar modelos existentes para a nossa solução, representem esses modelos os tipos de dados que um determinado modelo conterá ou simplesmente sub-modelos previamente desenvolvidos e validados que devem ser incluídos a determinado ponto da transformação. Para este efeito criamos a definição de importação na nossa linguagem.
Esta operação é feita de forma semelhante a uma qualquer regra de transformação. Temos então um *MatchModel* com uma referência directa para o modelo a importar. Neste padrão representamos qual a raiz ou sub-raiz que queremos capturar no modelo e indicamos através de uma ligação específica (*Import* na sintaxe concreta da linguagem) qual a nova raiz a utilizar no modelo de saída. Com isto serão copiados todas as entidades contidas na sub-árvore do modelo a importar para o modelo de saída e respectivas relações que estão totalmente contidas nessa

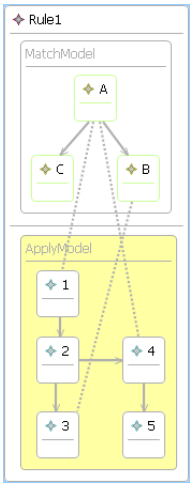

Figura 4.7: Restrição imposta através do histórico de uma entidade

sub-árvore.

4.6 Editor

O editor da linguagem foi desenvolvido utilizando as plataformas GMF e EuGENia descritos anteriormente nas secções 2.5 e 2.5.1 respectivamente. O editor encontra-se assim totalmente integrado na plataforma Eclipse assim como o próprio motor de transformação permitindo efectuar a aplicação das transformações sem a necessidade de uma mudança de ambiente de desenvolvimento, pois a transformação é feita de forma automática não gerando código editável pelo utilizador.

4.7 Sintaxe Concreta

Aqui explicamos todos os objectos e ligações presentes no editor da linguagem, ou seja, a sintaxe concreta da mesma e com a qual podemos efectivamente definir as regras de transformações de modelos.

4.7.1 Objectos

MetaModelId Um *MetaModelId* identifica o metamodelo de uma determinada fonte, seja ela uma camada (*Layer*) ou um ficheiro *FilePort*). Este elemento é obrigatório em todas as fontes e necessita do nome e pacote do metamodelo assim como o caminho para o ficheiro .ecore que o representa.

FilePort Um *FilePort* é utilizado para carregar um determinada modelo para o motor de transformação. Este modelo serão depois utilizado como fonte de uma camada ou de um *MatchModel*, sobre o qual será feito o processo de captura de padrões. Necessita de um *MetaModelId* com a informação relativa ao metamodelo do modelo que está a carregar e do caminho para o ficheiro xmi que contém o modelo.

Layer Uma camada (*Layer*), como descrita anteriormente é uma forma de dar alguma ordem à execução da transformação. A *Layer* vai conter u conjunto de regras, assim como um *MetaModelId*. O *MetaModelId* identifica o metamodelo utilizado pelo modelo de saída produzido por cada camada.

Rule Uma regra (*Rule*) define uma transformação entre os modelos de entrada e saída. Este elemento irá conter vários modelos de captura (*MatchModel*), responsáveis pelos padrões a encontrar no modelo de entrada, e um modelo de produção (*ApplyModel*), responsável pelos padrões a criar no modelo de saída de camda camada.

MatchModel O *MatchModel* é o modelo que vai definir os padrões da regra de transformação a capturar no modelo de entrada (referido anteriormente como modelo de captura). Os padrões serão compostos por classes (*AnyMatchClass* e *ExistsMatchClass*) e associações (*PositiveIndirectAssociation*, *PositiveMatchAssociation*) que devem estar em conformidade com o metamodelo de input. Este modelo utiliza o modelo de entrada definido como fonte da camada, a menos que exista uma ligação explicita para um *FilePort*. Os padrões serão capturados sobre o modelo de entrada definido, se existir definição explicita, o modelo vindo através da camada é ignorado.

ApplyModel O *ApplyModel* contém o padrão de produção que irá ser criado no modelo de saída. Este modelo é composto por classes (*ApplyClasses*)e associações (*ApplyAssociations*) que devem estar em conformidade com o metamodelo de saída definido na camada. Existe apenas um *ApplyModel* para cada regra.

AnyMatchClass A *AnyMatchClass* define uma meta-classe do metamodelo de entrada e captura todas as soluções presentes no modelo de entrada. Serão geradas soluções distintas para cada uma das ocorrências, podendo ser aplicado várias vezes o padrão de produção, dependendo da restante regra de transformação.

ExistsMatchClass A *ExistsMatchClass* define de igual modo uma meta-classe do metamodelo de entrada, no entanto captura apenas uma solução existente no modelo de entrada, gerando apenas uma solução ainda que existam várias possibilidades no modelo de entrada.

ApplyClass A *ApplyClass* define uma classe a ser produzida no modelo de saída. Apenas existe um tipo de *ApplyClass*. Cada classe pode ser produzida várias vezes no modelo de saída, dependendo para isso das soluções geradas pelo modelo *MatchModel*.

MatchAttribute Um *MatchAttribute* define um atributo do modelo de entrada e pode ser utilizado de duas formas. Pode ser utilizado como uma referência para a construção de um atributo de saída (*ApplyAttribute*), ou pode ser usado como restrição ao padrão de captura, restringindo as soluções encontradas àquelas que respeitem o valor definido no atributo.

Os valores do *MatchAttribute* podem ser de dois tipos, um *Atom* ou um *IsNull* que serão explicados adiante nesta secção.

ApplyAttribute O *ApplyAttribute* é uma entidade bem mais complexa do que o *MatchAttribute*. Uma vez que estamos a criar novos atributos, criamos uma pequena linguagem para definição de atributos, que inclui referências para outros atributos (de entrada ou outros de saída), operações entre atributos e pode funcionar como refinamento às restrições retroactivas *BackwardRestrictions*.

Esta linguagem define-se pela seguinte gramática:

TERM $\rightarrow$ REF | OPERATOR | Atom | WildCard
REF $\rightarrow$ AttributeRef | ClassRef
OPERATOR $\rightarrow$ TYPEOF | CONCAT
TYPEOF $\rightarrow$ AttributeRef
CONCAT $\rightarrow$ TERM TERM

Os *Atom*, *WildCard*, *AttributeRef* e *ClassRef* são os simbolos terminais desta gramática.

Atom Um *Atom* representa o valor de um determinado atributo num tipo de dados primitivo que deve corresponder ao tipo definido no metamodelo.

Wildcard O *WildCard* é utilizado para compor expressões com valores desconhecidos ou irrelevantes, para serem utilizadas nas restrições retroactivas.

AttributeRef O *AttributeRef* é o objecto da sintaxe concreta que irá funcionar como fonte da referência para outro atributo. Esta referência copia o valor do atributo para o qual a ligação irá apontar.

ClassRef Semelhante ao *AttributeRef*, este objecto referencia classes, copiando o seu nome, podendo ser útil para referenciar tipos de atributos complexos.

IsNull Este objecto foi adicionado como forma de poder verificar se um determinado atributo de uma classe de captura é nulo ou não, funcionando assim como restrição ao padrão de captura.

4.7.2 Ligações

ApplyAssociation A *ApplyAssociation* representa as ligações entre duas *ApplyClasses* e necessita estar em conformidade com o metamodelo de saída para que seja processada a transformação da regra respectiva.

AttributeRef A ligação *AttributeRef* indica qual o atributo a copiar para o objecto com o mesmo nome.

ClassRef A ligação *ClassRef* indica qual a classe cujo nome será copiado para o local onde se encontra o objecto com o mesmo nome desta ligação.

ExplicitSource Esta ligação indica uma fonte explícita para um determinada *MatchModel*. Esta fonte pode ser apenas do tipo *FilePort*.

Import A ligação *Import* indica qual a nova raiz da sub-árvore a capturar num determinada modelo de entrada.

PositiveBackwardRestriction Um *PositiveBackwardRestriction* é a ligação que vai indicar onde será aplicada a característica de restrição retroactiva. Tem necessariamente como origem uma *MatchClass* e como destino uma *ApplyClass*.

NegativeBackwardRestriction Esta ligação é análoga à anterior, indicando que quer uma qualquer *ApplyClass* já existente mas que não foi criada a partir de uma determinada *MatchClass*.

PostiveIndirectAssociation A *PositiveIndirectAssociation* representa a ligação de definição da funcionalidade de associações indirectas. Tanto a origem como o destino desta ligação tem obrigatoriamente que ser uma *MatchClass*.

NegativeIndirectAssociation Funciona de forma inversa à ligação anterior. Indica que não pode existir um caminho indirecto entre duas classes.

PositiveMatchAssociation Representa uma associação directa entre duas *MatchClasses*. Tanto a origem como o destino desta ligação tem obrigatoriamente que ser uma *MatchClass*.

NegativeMatchAssociation Funciona como uma NAC sobre uma associação indirecta. Indica que não pode existir determinada associação no modelo de entrada. Tanto a origem como o destino desta ligação tem obrigatoriamente que ser uma *MatchClass*.

PreviousSource A ligação *PreviousSource* indica qual a fonte de uma camada. Isto é necessário para determinar o fluxo de execução da transformação. Têm como origem uma camada, enquanto o destino pode ser outra camada ou um ficheiro. Cada camada tem obrigatoriamente que ter uma ligação destas, caso contrário não será processada.

4.8 Semântica

A semântica da linguagem proposta foi implementada sobre a linguagem Java, com recurso ainda a predicados Prolog. O motor Prolog é utilizado para gerar soluções possíveis para os padrões definidos nas regras de transformação, ou seja, para cada padrão de captura, são gerados todas as soluções existentes no modelo de entrada sendo estas posteriormente filtradas em função de outras restrições presentes no padrão, por exemplo condições sobre atributos.

A execução de regras é feita de forma não-determinística. Deste modo pretende-se reduzir o número de possibilidades para a definição das regras de transformação para um determinado caso. Com isto pretende-se reduzir a liberdade de definição de regras, o que por um lado reduz o poder de expressividade do utilizador, mas possibilita o utilizador ganhar mecanismos de utilização e uma maior adaptabilidade à própria ferramenta de transformação. Isto devido às tarefas semelhantes serem executadas habitualmente da mesma forma.

A imagem 4.8 apresenta um esquema de execução do motor de transformações. Começa então por carregar para memória o modelo de transformação que contém todas as camadas e respectivas regras de transformação. Posteriormente, a cada camada de regras são verificadas as suas precedentes de forma a garantir que esta possa ser processada, pois cada camada necessita dos modelos processados pelas fontes anteriores sejam elas outras camadas (*Layer*) ou ficheiros de entrada (*Fileport*). Caso esses requisitos não estejam ainda cumpridos, ou seja, as fontes anteriores não estejam ainda processadas, processamos estas em primeiro lugar de forma a poder garantir uma correcta aplicação das camadas, e correspondentes regras, pela ordem definida no modelo de transformação.

No fluxograma da figura 4.9 é mostrado o fluxo de execução das regras de transformação dentro de cada camada. Em cada camada é seleccionada uma regra de transformação e verificado se existem soluções possíveis no modelo de entrada. Caso existam, é produzido o padrão de produção (*ApplyModel*) no modelo de saída da camada correspondente. Após processar todas as regras da camada, verifica-se se existem mais soluções possíveis ainda à espera de processar, caso isto se verifique é executada uma nova passagem pelas regras de transformação consumindo uma outra solução ainda não tratada.

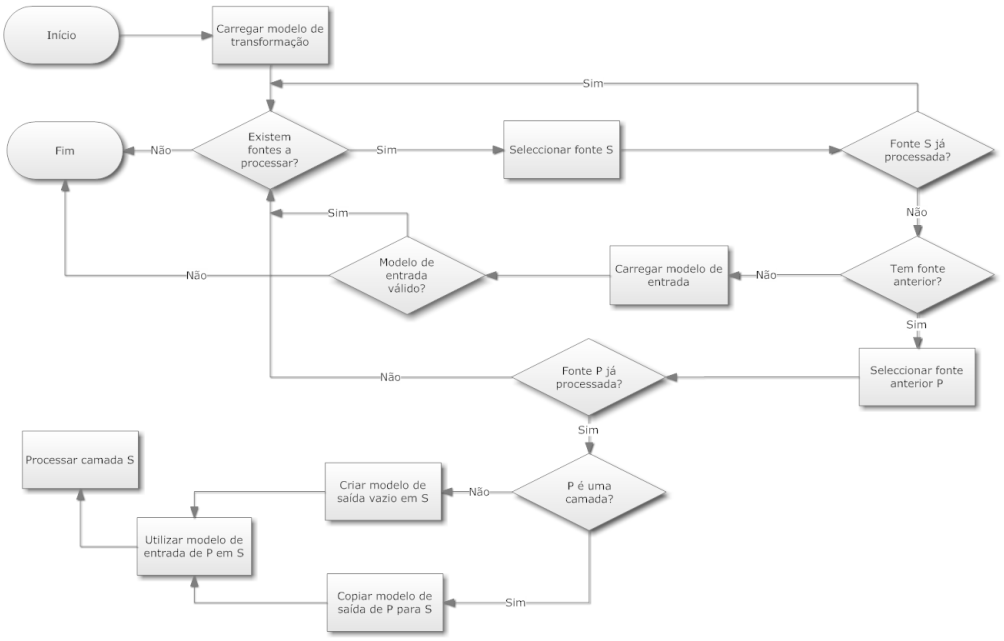

Figura 4.8: Esquema de execução do motor de transformação.

4.8.1 Definição da Transformação

De forma a formalizar a semântica da nossa linguagem temos que observar os modelos como um conjunto de nós e vértices representando um grafo de descrição directa.

Em relação ao modelo de transformação, como existe uma separação sintáctica entre o modelo de captura (respeitante ao modelo de entrada - *Match*) e o modelo de aplicação (respeitante ao modelo de saída - *Apply*) é necessário efectuar essa distinção de forma análoga às regras de reescrita de grafos por LHS (*MatchModel* na nossa linguagem) e RHS (*ApplyModel na nossa linguagem*). No entanto, devido à produção de um modelo de saída de forma incremental, esta notação não será exactamente semelhante à notação de reescrita de grafos.

Definição *Match-Apply Model*

Consideramos um *Match-Apply Model* (MAM) um modelo de agregação de alguns dos elementos pertencentes à transformação, nomeadamente o modelo de entrada sobre o qual serão encontradas as soluções de transformação(*Match*), o modelo de saída onde será construído o modelo final através das regras definidas(*Apply*) e as respectivas ligações de restrições retroactivas.

Definição *Regra de Transformação*

Uma regra de transformação é um *Match-Apply Model* com a particularidade de nesta fase se introduzir o conceito de relações indirectas. O conjunto de todas as regras de transformação é denominado de TR_t^s. As relações indirectas são apenas permitidas no padrão de captura mas

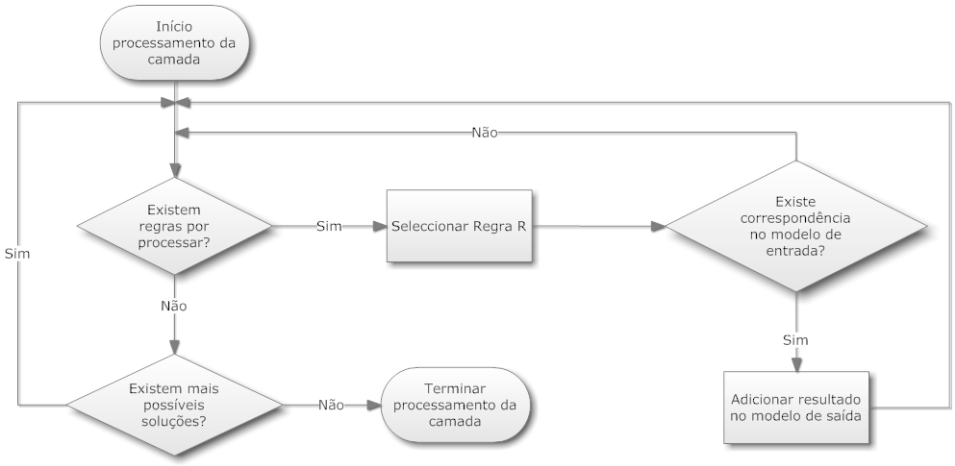

Figura 4.9: Esquema de execução das regras dentro de cada camada.

não no de produção, pois no padrão de produção está a ser definida a construção do modelo de saída e não é possível construir relações indirectas de forma tão abstracta.

Definição *Layer, Transformation*

Uma camada é um conjunto finito de regras de transformação $tr \subseteq TR_l^s$. O conjunto de todas as camadas é chamado *Layer*. Uma transformação é um conjunto finito de camadas $[l_1 :: l_2 :: \ldots :: l_n]$ onde $l_k \in Layer$ e $1 \leq k \leq n$. O conjunto de todas as transformações é denominado $Transformation$.

Definição Função *Match*

A função *match* é a responsável por encontrar todas as ocorrências de um determinado padrão definido pela regra de transformação tr no modelo de entrada. Esta função gera um conjunto de soluções possíveis utilizadas pela função *transform*.

Definição Função *Transform*

A função *transform* será então a responsável pela aplicação do modelo de produção a cada solução de uma determinada regra de transformação, encontradas pela função *match*. Esta função adiciona a cada solução o respectivo padrão de produção definido pela regra de transformação assim como informação sobre o histórico da transformação. Em relação ao histórico, são criadas ligações entre todos os elementos do padrão de captura para todos os elementos do padrão de produção que estejam a ser criados nesta altura, ou seja, os elementos existentes no padrão de produção que já tenham histórico associado são ignorados. Termina quando todas as soluções geradas pela função *match* já estiverem tratadas devolvendo o modelo com a união de todas as soluções tratadas.

4.8.2 Semântica da Transformação

Definição Semântica de *Layer Step*

Seja $l \in Layer$ uma camada. A relação $layer\ step \overset{layerstep}{\rightarrow} \subseteq MAM_t^s \times TR_t^s \times MAM_t^s$ é definida como:

$$\frac{}{< m, m', \emptyset > \xrightarrow{layerstep} m \sqcup m'}$$

$$\frac{tr \in l,\ transform_{tr}(m) = m'',\ < m, m' \sqcup m'', l \backslash \{tr\} > \xrightarrow{layerstep} m'''}{< m, m', l > \xrightarrow{layerstep} m'''}$$

onde $\{m, m', m''\} \subseteq MAM_t^s$ são *match-apply models*.

Sendo tr uma regra de transformação pertencente uma determinada camada l. É aplicada a função $transform_t r$ sobre o MAM m executando a transformação da regra e consequentemente gerando um modelo de saída intermédio m'' através da aplicação da transformação dessa regra. Após o processo de transformação a regra processada é retirada ao conjunto de regras da camada actual e processado o restante conjunto de regras pelo mesmo processo. A utilização deste modelo intermédio deve-se ao facto das restrições retroactivas geradas numa camada, só poderem ser utilizadas na camada seguinte. O processamento da camada termina quando todas as regras estiverem processadas, ou seja, quando o conjunto de regras da camada estiver vazio.

Definição Semântica de *Transformation Step*

Seja $[l :: R] \in TR_t^s$ uma *Transformation*, onde $l \in Layer$ é uma camada e R uma lista. A relação $transformation\ step \overset{trstep}{\rightarrow} \subseteq MAM_t^s \times TR_t^s \times MAM_t^s$ é definida como:

$$\frac{}{m, [] \xrightarrow{trstep} m}$$

$$\frac{< m, \emptyset, l > \xrightarrow{layerstep} m'',\ m'', R \xrightarrow{trstep} m'}{m, [l :: R] \xrightarrow{trstep} m'}$$

onde $\{m, m', m''\} \subseteq MAM_t^s$ são *match-apply models*.

A cada passo da transformação é processada uma camada l, ou seja, são processadas um conjunto de regras de transformação. A cada camada é construído um modelo de saída intermédio contido em m''. Após o processamento de cada camada, esta é retirada da lista de camadas a processar dando início ao processamento da seguinte, até não existirem mais camadas a processar no fluxo de execução da transformação. Quando não existem mais camadas a processar obtemos o modelo de saída final contido em m'.

Definição Transformação de Modelos

Seja $m_s \in MODEL^s$ e $m_t \in MODEL^t$ modelos e $t \in Transformation_t^s$ uma regra de transformação. Uma transformação de modelo $\xrightarrow{tr}: MODEL^s \times Transformation_t^s \times MODEL^t$ é definida como:

$$m_s, t \xrightarrow{tr} m_t \Leftrightarrow \langle V, E, T, m_s, \emptyset, \emptyset \rangle, t \xrightarrow{trstep} \langle V, E, T, m_s, m_t, Bl \rangle$$

De uma forma geral uma transformação de modelos é dada por um modelo de entrada m_s que através da execução da transformação definida em t dá origem ao modelo de saída m_t. Este modelo de saída é construído de forma incremental através da aplicação sucessiva de etapas de transformação definidas anteriormente.

4.9 Exemplo de Transformação

Para se compreender melhor a linguagem e respectiva ferramenta de transformação, apresentamos em seguida um modelo de transformação de um modelo de diagramas de classes para um modelo relacional. O exemplo utilizado foi adaptado de [32] sendo de seguida explicada cada uma das camadas de transformação definidas.

4.9.1 Camada 0

A figura 4.10 apresenta a camada inicial do nosso modelo de transformação. Esta camada tem a particularidade de utilizar um *FilePort* como fonte. O *FilePort* é responsável pela leitura do modelo de entrada (que contém um modelo de diagrama de classes) e passa-o como modelo de entrada para esta camada.

Nesta camada apenas criamos a raiz do nosso modelo de saída. Uma vez que queremos uma raiz única para o nosso modelo, começamos pela criação de um **ERModel** por transformação de um **Diagram** do modelo de entrada.

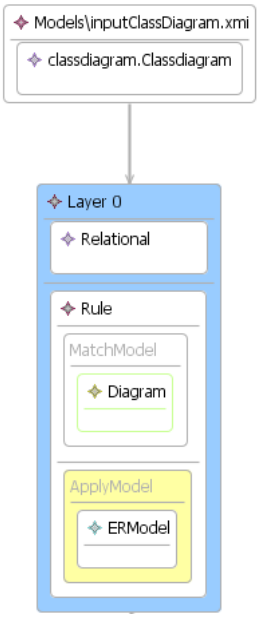

Figura 4.10: Camada 0 do exemplo de transformação de modelos

4.9.2 Camada 1

Após o carregamento para memória do modelo de entrada e com a raiz criada no modelo de saída, podemos começar a descrever as regras de transformação entre diagramas de classe e modelos relacionais. Na camada 1, temos 2 regras como mostrado na na figura 4.11. A primeira, encontra classes definidas no modelo

de entrada e que são persistentes, e cria tabelas com o
mesmo nome no modelo de saída conectando-os com o objecto raiz **ERModel**.

Precisamos também, de criar no modelo de saída os tipos disponíveis no modelo de entrada. Assim, com outra regra, vamos capturar todos os tipos primitivos presentes no modelo de diagrama de classe e para cada um, criar um tipo correspondente no modelo de saída e conectá-los com a raiz criada na camada anterior. Finalmente, capturamos uma instância de cada classe do modelo de entrada, usando a *ExistsMatchClass*, e criamos um tipo respectivo no modelo de saída. Neste caso particular, se usássemos o *AnyMatchClass*, estaríamos a criar um novo tipo para cada instância de cada classe o que geraria redundância de tipos no modelo de saída.

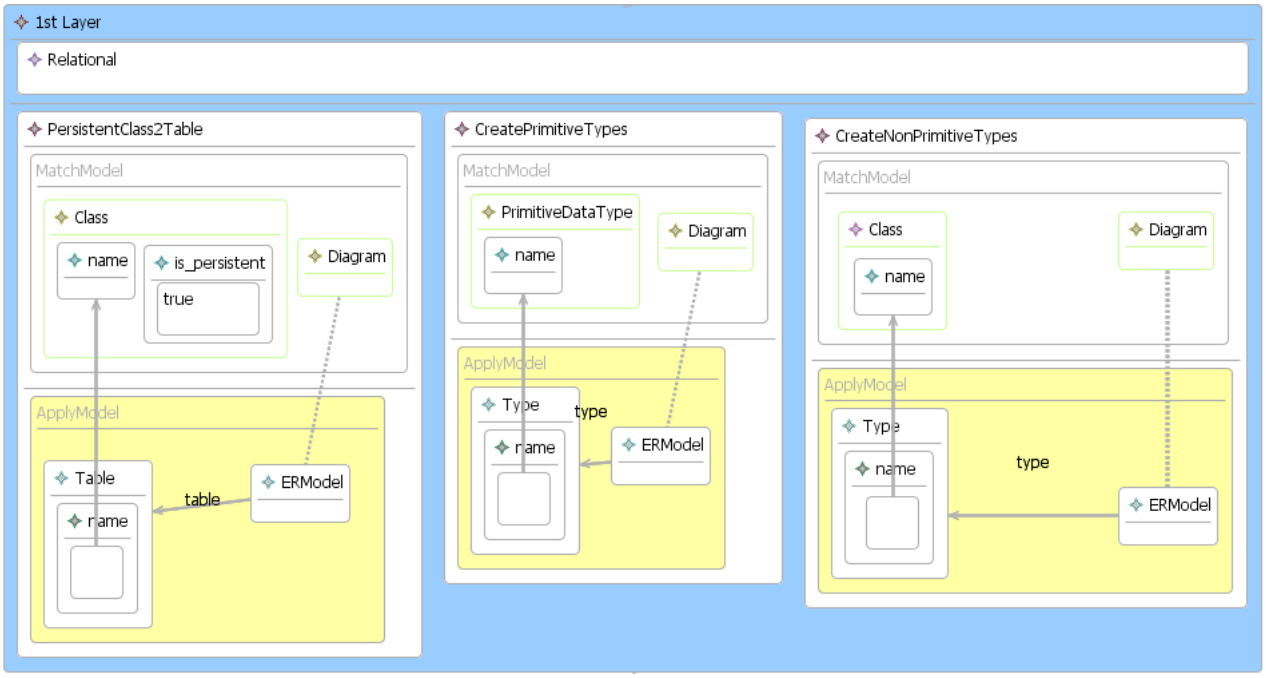

Figura 4.11: Camada 1 do exemplo de transformação de modelos

4.9.3 Camada 2

Na camada 2, existem também duas regras. A primeira, procura por todos os atributos (*Attribute*) em classes previamente transformadas e transforma cada solução em uma coluna (*Column*) no modelo de saída e liga-as com seu tipo (*Type*) correspondente também criado anteriormente. A outra regra, procura por classes não persistentes ligadas a uma persistente através de uma associação (*Association*) e cria uma *Column* para cada um dos seus atributos, mas neste caso, eles são ligados à tabela(*Table*) criada através da classe persistente.

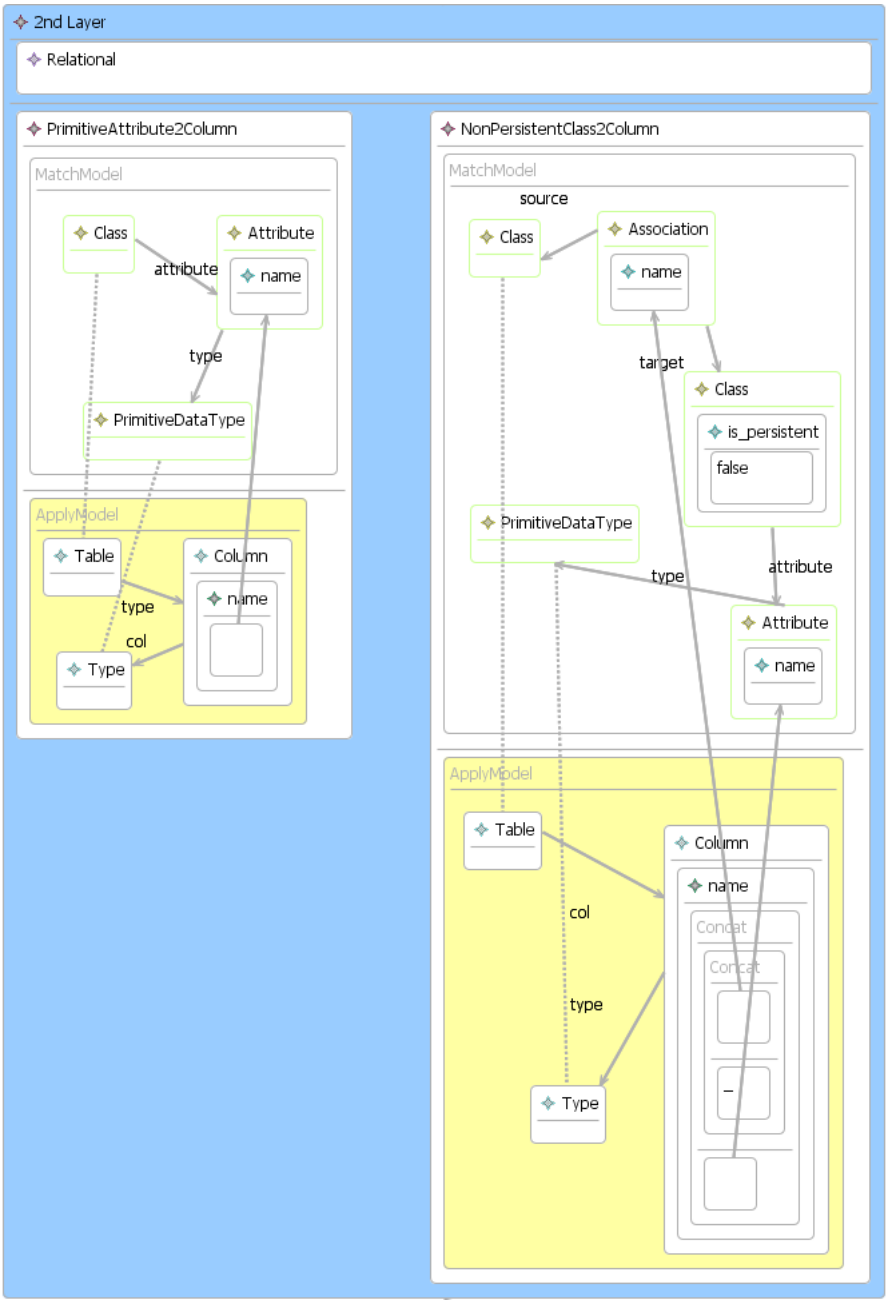

Figura 4.12: Camada 2 do exemplo de transformação de modelos

4.9.4 Camada 3

A esta altura, teremos as classes e os atributos base transformados num modelo relacional. Então precisamos especificar as transformações menos triviais.

Vamos começar com atributos que têm tipos primitivos, ou seja, atributos que representam um objecto de outra classe persistente. Isto é definido na primeira regra desta camada, combinando os atributos que têm uma classe como tipo e criar uma coluna correspondente que será uma chave externa da tabela que a contém, referenciando assim a tabela criada através da classe que representa o tipo do atributo.

A 2ª regra desta camada captura as associações entre duas classes persistentes e cria uma chave externa na tabela de origem com um nome composto. A 3ª regra procura por 2 classes persistentes que estão conectadas através de uma não-persistente. Neste caso, vamos também criar uma chave externa na tabela de origem com um nome composto, mas agora o nome irá considerar mais valores.

4.9.5 Camada 4

Na última camada deste exemplo simples, acabamos definindo as chaves necessárias para as tabelas já criadas. Fazemos isto por último, pois apenas neste momento sabemos que todas as tabelas e colunas estão criadas, porque as dependências retroactivas apenas estão disponíveis para os elementos criados até a camada anterior.

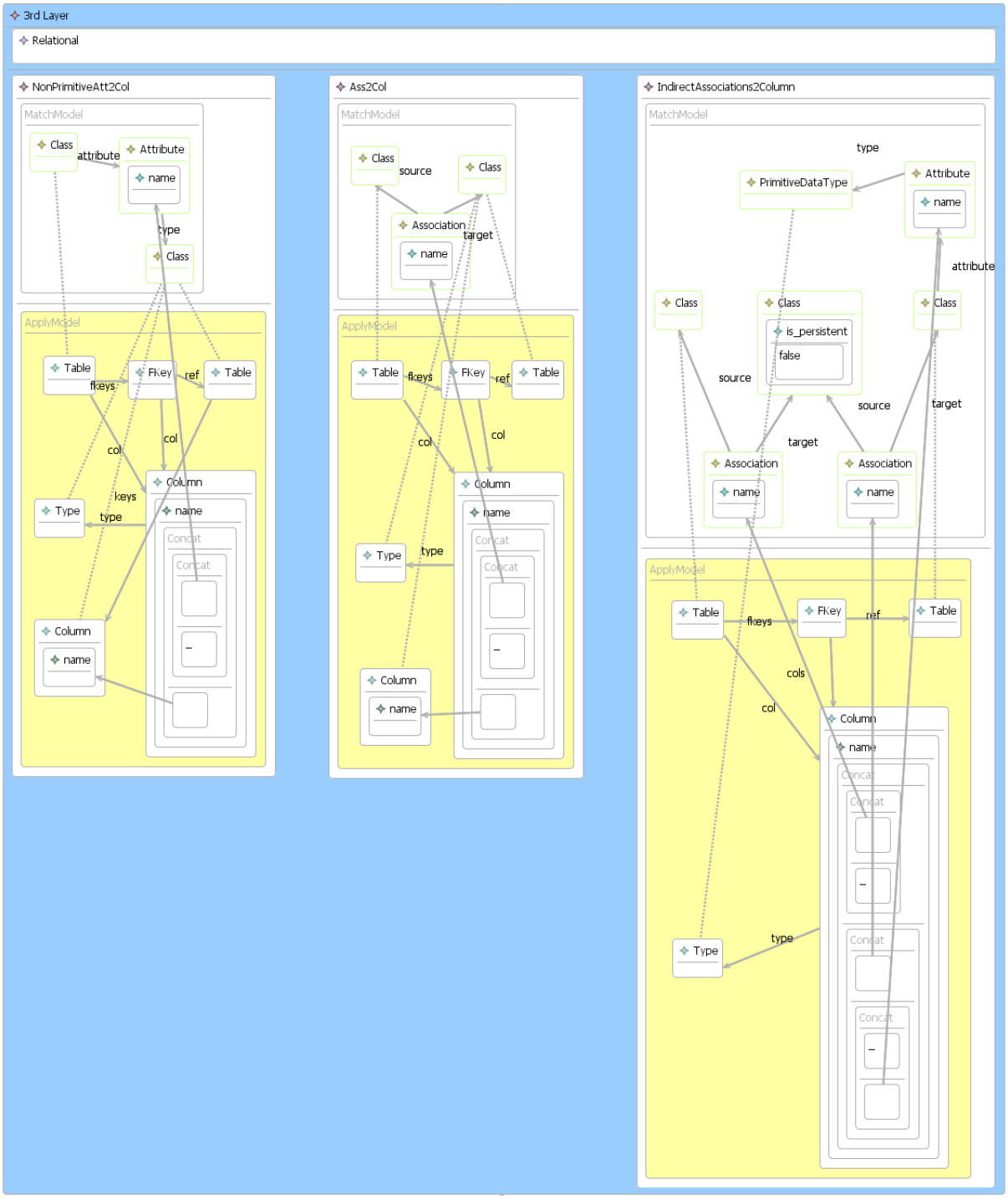

Figura 4.13: Camada 3 do exemplo de transformação de modelos

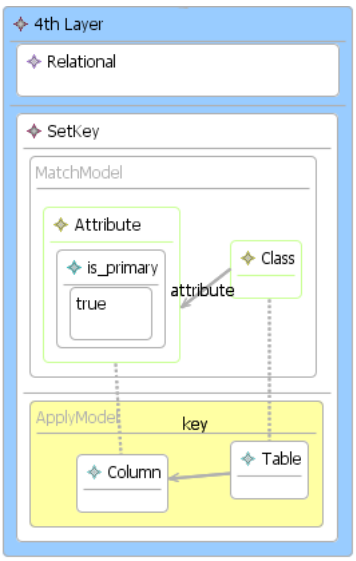

Figura 4.14: Camada 4 do exemplo de transformação de modelos

5

Validação

Para avaliarmos a nossa ferramenta, efectuamos dois estudos de modo a testar convenientemente diferentes aspectos da ferramenta e subsequente linguagem de transformação desenvolvida, nomeadamente através de um teste à usabilidade e à expressividade que serão descritos em seguida.

5.1 Avaliação Experimental

Devido à nossa ferramenta de transformação ser também uma LDE, foi aplicada uma avaliação experimental de modo a obter *feedback* por parte do público alvo e assim avaliar o seu sucesso. Esta avaliação consistiu na criação de um cenário de teste de modo a perceber o que foi atingido e o que poderia ser melhorado ao nível da interacção pessoa máquina nesta ferramenta. De forma a obter resultados mais relevantes para a análise da usabilidade da ferramenta desenvolvida em relação às existentes, utilizamos também a ferramenta Moflon como plataforma de comparação, segundo os mesmos critérios. Esta ferramenta foi escolhida devido à sua interface gráfica de definição de regras, obtendo por isso uma usabilidade que se julgaria próxima da ferramenta desenvolvida.

5.1.1 Factores de sucesso

De forma a podermos analisar o êxito do DSLTranslator foi necessário identificar um variado conjunto de factores de sucesso de LDEs. Estes factores de sucesso foram encontrados através de trabalhos já efectuados na área da experimentação de LDEs nomeadamente em [41] e

[42]. Estes factores de sucesso foram obtidos considerando especificamente o desenvolvimento de LDEs, não considerando assim factores de índole geral.
Os factores utilizados são os seguintes:

Aprendizagem (L) Os utilizadores da linguagem desenvolvida têm que aprender uma nova linguagem o que requer tempo e esforço, daí este ser um factor de sucesso, ou não, de uma nova linguagem. Uma linguagem que consiga uma curva de aprendizagem suave terá uma mais valia na adopção da mesma por parte dos utilizadores,assim como, facilita uma futura actualização de utilizadores antigos à medida que o domínio muda e a LDE evolui naturalmente.

Familiaridade (F) Devido ao facto de os utilizadores necessitarem de aprender uma nova linguagem, o facto de estes reconhecerem aspectos familiares aos que já conhecem pode influenciar a aprendizagem e futura adopção da mesma como ferramenta de trabalho.

Usabilidade (U) A interacção das ferramentas associadas à linguagem, nomeadamente do seu editor com o utilizador, deve ser fácil e intuitiva de modo a que se evitem erros derivados à utilização das mesmas e deste modo evitar perdas de produtividade.

Eficiência (E) Uma LDE pretende reduzir esforço e tempo de desenvolvimento, para isso é importante automatizar grande parte do processo de desenvolvimento contribuindo também para uma menor percentagem de erros.

Expressividade (EX) Uma LDE é por norma uma linguagem que limita a expressividade a um determinado domínio para assim obter ganhos em produtividade e eficiência, no entanto é necessário verificar se a linguagem desenvolvida contém um nível de expressividade suficiente para o contexto em que esta deve ser utilizada.

5.1.2 Utilizadores

Para a escolha dos utilizadores a utilizar nesta avaliação experimental foi necessário ter em conta que esta linguagem se trata de uma linguagem de transformação de modelos. Optamos então por separar os utilizadores em dois grupos, sendo estes "utilizadores experientes" ou "utilizadores não experientes". Todos os utilizadores, no entanto, teriam que ter uma base de conhecimento relativo alargado ao nível de conhecimentos informáticos e também de programação de modo a estarem identificados com as ferramentas e conceitos utilizados na execução do cenário proposto.

As diferenças entre os grupos são então na experiência de cada utilizador no uso de ferramentas de transformação de modelos. Deste modo é possível avaliar a ferramenta em relação às funcionalidades esperadas por parte dos utilizadores com algum conhecimento de outras ferramentas assim como avaliar o esforço necessário para novos utilizadores capturarem os conceitos introduzidos.

No total foram convidados a efectuar esta avaliação 16 pessoas, as quais sabíamos que teriam conhecimentos de programação com um mínimo de 3 anos de experiência a nível académico. Destas 16 pessoas responderam-nos 12 sendo que 6 nunca tiveram contacto com ferramentas de transformação e as restantes 6 já tinham contactado com pelo menos 1 ferramenta de transformação.

Embora o número de utilizadores seja bastante reduzido e não obtenha assim qualquer relevância estatística, segundo um grande especialista em usabilidade Jakob Nielsen, este é um número razoável[1] para a detecção da grande maioria dos erros de utilização de uma ferramenta[43].

5.1.3 Cenário de avaliação

O cenário apresentado para este estudo é bastante simples, pois queríamos deixá-lo acessível não só aos utilizadores experientes em transformações mas também para aqueles que as encontram pela primeira vez. Este cenário utiliza dois metamodelos muito restrito de UML (diagramas de classes) e Java, designados *SimpleUML* e SimpleJava sendo ambos fornecidos. Como o principal objectivo deste estudo é avaliar a usabilidade da nossa ferramenta, são indicadas as transformações a indicar com as respectivas condições a manter no modelo de saída, para que o utilizador possa traduzi-los na nossa sintaxe concreta.

5.1.3.1 Metamodelos

SimpleUML

O metamodelo *SimpleUML* abrange as entidades base dos diagramas de classes com pacotes, classes, operações e atributos descritos pela OMG e simplificado apenas para fins de transformação.

SimpleJava

O *SimpleJava* é a nossa linguagem alvo e consiste num *JavaModel* que é o elemento raiz e irá suportar todo o nosso modelo. O *JavaModel* contém pacotes que contém *JavaClasses* e *PrimitiveTypes*. Cada *JavaClass* pode conter *Fields* e *Methods* que também pode ter outros *Fields*, sendo neste contexto os argumentos dos métodos. Finalmente, campos e métodos têm a referência a um tipo que pode ser um *PrimitiveType* ou um *JavaClass*.

5.1.3.2 Especificação das Regras

Aqui enunciamos as regras de transformação que se pretende implementar neste cenário de avaliação para transformar um modelo UML num modelo Java, sendo elas as seguintes:

- Para cada instância *Package* em UML, tem que ser criado uma instância *Package* em Java.

  Os seus nomes têm que corresponder.

- Para cada instância de *Class* em UML, tem que ser criado uma instância *JavaClass*.

[1] http://www.useit.com/alertbox/20000319.html

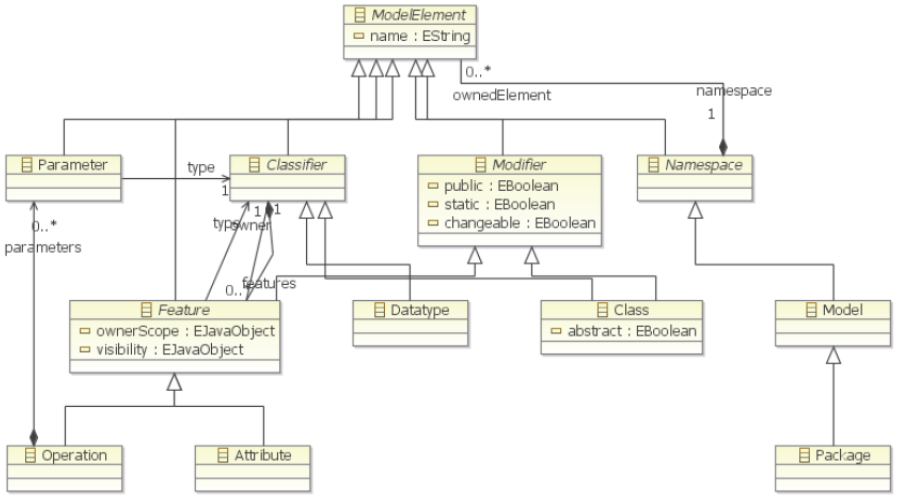

Figura 5.1: Metamodelo da linguagem SimpleUML utilizada no cenário de avaliação

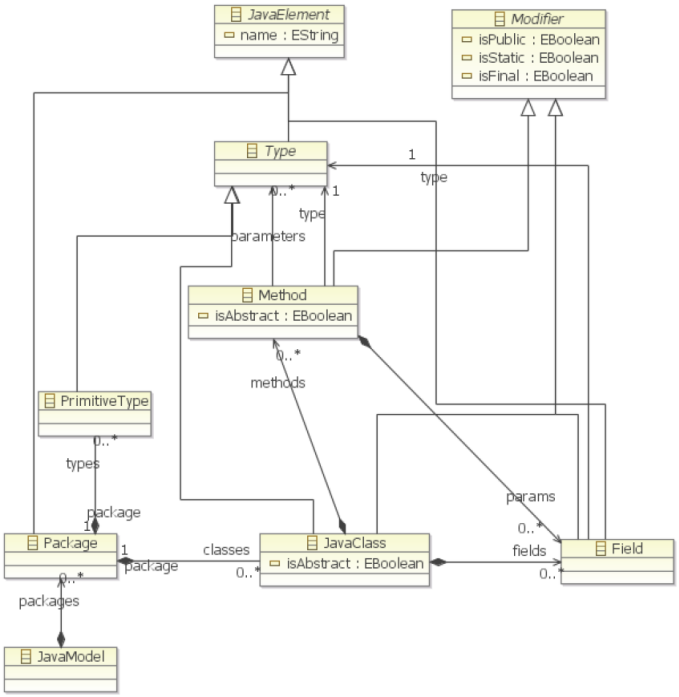

Figura 5.2: Metamodelo da linguagem SimpleJava utilizada no cenário de avaliação

Os seus nomes têm que corresponder.

A referência para o *Package* tem que corresponder.

Os modificadores têm que corresponder.

- Para cada instância *DataType* em UML, tem de ser criado uma instância *PrimitiveType* em Java.

  Os seus nomes têm que corresponder.

  A referência para o *Package* tem que corresponder.

- Para cada instância de *Attribute* em UML, tem de ser criado uma instância de *Field* em Java.

  Os seus nomes têm que corresponder.

  Os seus tipos têm de corresponder.

  Os modificadores têm que corresponder.

  As classes a que pertencem têm que corresponder.

- Para cada instância *Operation* em UML, tem de ser criado uma instância de *Method* em Java.

  Os seus nomes têm que corresponder.

  Os seus tipos têm de corresponder.

  Os modificadores têm que corresponder.

  As classes a que pertencem têm que corresponder.

- Para cada instância de *Parameter* em UML, tem de ser criado uma instância de *Field* em Java.

  Os seus nomes têm que corresponder.

  Os seus tipos têm de corresponder.

  A *Operation*/*Method* proprietário tem que corresponder.

5.1.4 Questionários de avaliação

Cada questão do questionário final de avaliação relaciona-se com cada um destes factores de desenvolvimento de uma linguagem de domínio específico, pois é essencial que as questões estejam directamente relacionadas com os objectivos desta avaliação. Foram também adicionadas perguntas de resposta aberta de modo a obter maior informação sobre a opinião dos utilizadores e assim aprofundar os resultados obtidos através dos mesmos.

Em seguida apresentamos uma visão geral do questionário utilizado discutindo os factores de sucesso e respectivas questões utilizadas para medi-los. A tabela 5.1 apresenta o questionário sobre o cenário apresentado enquanto na tabela 5.2 é mostrado um questionário de âmbito mais geral sobre a linguagem desenvolvida e respectiva ferramenta de edição. Estes questionários fornecem uma visam geral sobre o questionário utilizado, não sendo no entanto a versão completa do mesmo. A versão completa do questionário utilizado nesta avaliação encontra-se no anexo A adaptada de [42].

O questionário está dividido em duas partes, em que uma está directamente relacionada com o cenário utilizado e a outra incide sobre a opinião dos utilizadores na utilização da ferramenta de uma forma mais genérica. O questionário termina com uma zona destinada aos utilizadores experientes, de forma a que possam expressar-se sobre as diferenças, boas ou más, entre a nossa proposta e as já utilizadas anteriormente.

5.1.4.1 Questionário sobre o cenário

Factor	**Questão**
	Aprendizagem
L3	Com que frequência teve necessidade de consultar a documentação?
L4	Com que frequência efectuou questões ao supervisor?
	Facilidade de Utilização
U2	Que nível de confiança sentiu durante a execução do cenário?
U3	Com que frequência se confundiu durante a execução do cenário?
U5	Quão mentalmente exigente foi o cenário? O que achou mais complicado de perceber ou executar?
	Eficiência
EF1	O que achou da correcção do cenário aplicado?
	Expressividade
EX1	Que achou da extensão do cenário realizado?

Tabela 5.1: Questionário sobre o cenário de avaliação

5.1.4.2 Questionário final

Factor	**Questão**
	Conhecimento do Utilizador
B1	Tem conhecimentos de programação? Como os classifica? Há quantos anos programa?

B2	Com que frequência utiliza ferramentas para modelação de LDEs? Por quanto tempo utilizou-a? Apreciou a experiência? Para que foi utilizado?
	Aprendizagem
L1	Com que facilidade aprendeu os conceitos?
L2	Quão útil foi o exemplo fornecido?
	Familiaridade
F1	Como identificou os símbolos que representam os conceitos? Qual achou inadequado?
F2	Como identificou o texto que representam os conceitos? Qual achou inadequado?
F3	Com que frequência cometeu erros devido à semelhança entre símbolos?
F4	Com que frequência cometeu erros devido ao vocabulário ambíguo?
	Facilidade de Utilização
U1	O que pensa da ferramenta?
U4	Qual a dificuldade sentida para aplicar alterações?
	Eficiência
EF2	O resultado reflecte o que estava à espera?
	Expressividade
EX2	Com que frequência não conseguiu expressar o que pretendia? Porquê?
	Impressões Gerais
G1	Qual a apreciação global da linguagem proposta?
G2	Que mudanças ou melhorias propõe para a ferramenta?
G3	Acredita que é uma mais-valia em comparação com outras? Porquê?
G4	Sentiu-se mais produtivo do que com outra linguagem de programação? Porquê?

Tabela 5.2: Questionário final da avaliação proposta

5.1.5 Execução da avaliação

O teste de avaliação ocorreu de forma assíncrona, sendo que o questionário foi disponibilizado online assim como todas as instruções para executar o cenário proposto. Isto tornou mais fácil a execução da avaliação, pois esta foi feita segundo a disponibilidade de cada utilizador. Foi fornecido também o manual da ferramenta, com um exemplo de transformação incluído de forma a introduzir os utilizadores à nova linguagem com que se depararam. Esta opção invalida ainda a possível ajuda dada pela equipa de desenvolvimento que poderia influenciar os próprios resultados de cada utilizador.

5.1.6 Resultados

Nesta secção apresentamos os resultados da avaliação devidamente separados pelos factores de sucesso identificados anteriormente.

5.1.6.1 Aprendizagem

Pudemos observar que os utilizadores experientes não tiveram grandes dificuldades na apreensão dos conceitos das linguagens utilizadas, tanto para DSLTrans como para Moflon. No entanto, como visível pela figura 5.3, os utilizadores não experientes tiveram uma maior facilidade a aprender os novos conceitos em DSLTrans do que em Moflon. Talvez o facto deste ultimo grupo de utilizadores não estar tão familiarizado com as ferramentas de transformação leve a uma maior dificuldade na aprendizagem da ferramenta Moflon, levando-nos a concluir que o DSLTrans introduz os conceitos da sua linguagem de forma mais simples e compreensível.

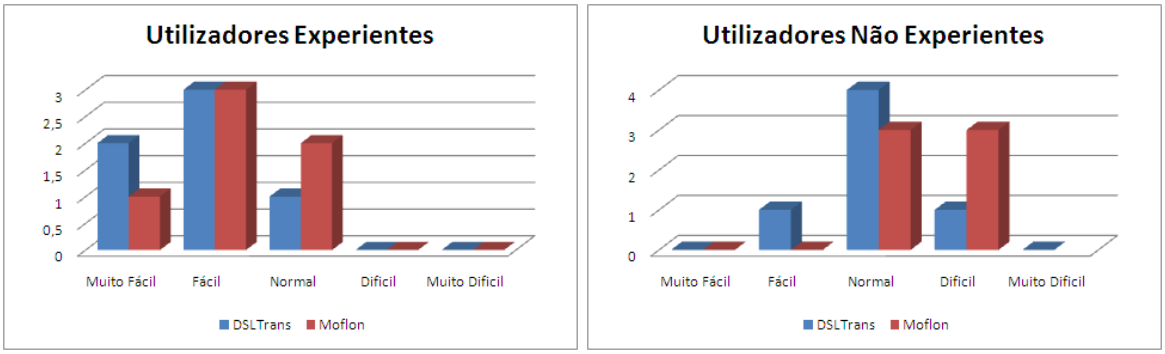

Figura 5.3: Resultados da questão L1 - Com que facilidade aprendeu os conceitos?

5.1.6.2 Familiaridade

Neste factor de desempenho da linguagem os utilizadores não diferenciaram de forma evidente as duas ferramentas em análise, embora os utilizadores não experientes indiquem um maior número de erros na ferramenta Moflon devido à semelhança de símbolos como apresentado na figura 5.4.

5.1.6.3 Usabilidade

A usabilidade da ferramenta, sendo uma característica fundamental no nosso trabalho, assume uma importância elevada nesta avaliação nomeadamente através dos utilizadores experientes pois estes conhecem os ambientes de transformação existentes e têm uma ideia mais concreta sobre o que existe e o que podia ser melhorado. Desta forma obtivemos uma resposta

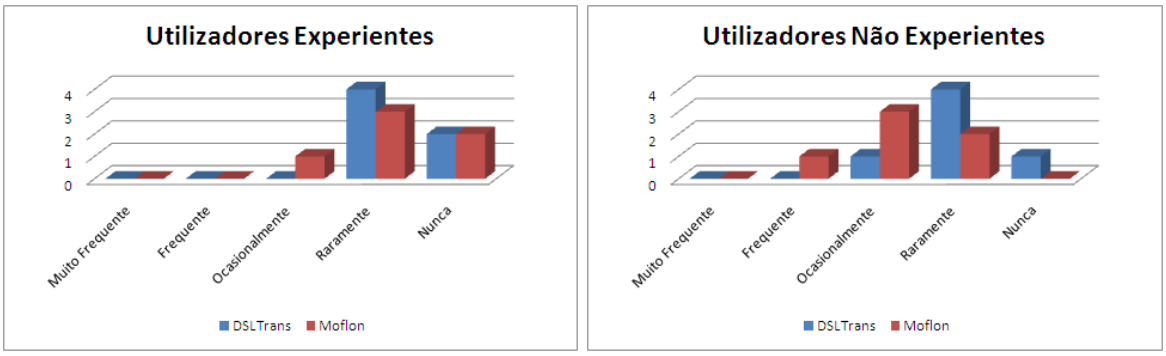

Figura 5.4: Resultados da questão F3 - Com que frequência cometeu erros devido à semelhança entre símbolos?

positiva por parte de todos os utilizadores, ainda que o facto de os utilizadores não experientes não conhecerem a realidade das ferramentas de transformação mais utilizadas possa ter contribuído para uma menor satisfação da usabilidade assim como mostra os gráficos da figura 5.5. Este resultado é ainda mais satisfatório se considerarmos que os utilizadores experientes consideram a ferramenta DSLTrans mais usável do que a Moflon.

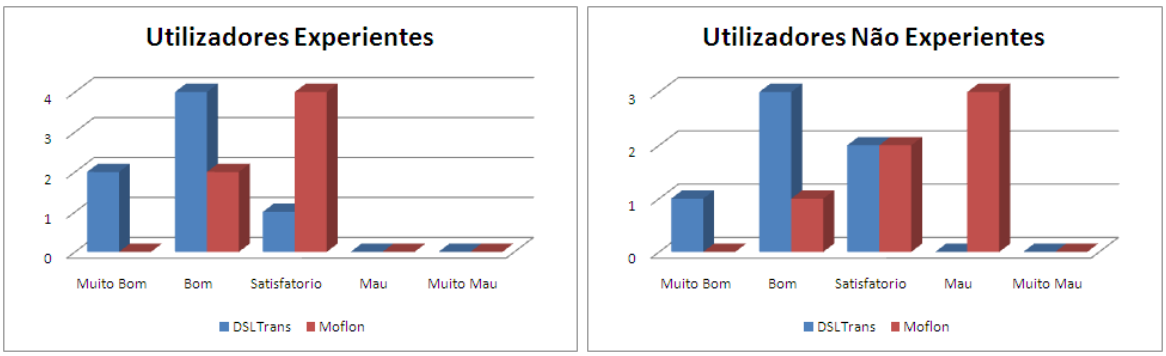

Figura 5.5: Resultados da questão U1 - O que pensa da ferramenta?

5.1.6.4 Eficiência

Neste aspecto, a eficiência do processo de transformação é reconhecido pela maioria dos utilizadores como demonstra a figura 5.6. A maior satisfação dos utilizadores em relação ao DSLTrans neste caso estará, essencialmente, relacionada com o facto da execução da transformação ser feita de forma simples, sem a necessidade de geração de código e migração desse código para outros interpretadores/ferramentas para que se obtenha o resultado pretendido, contrariamente ao que acontece com a ferramenta Moflon.

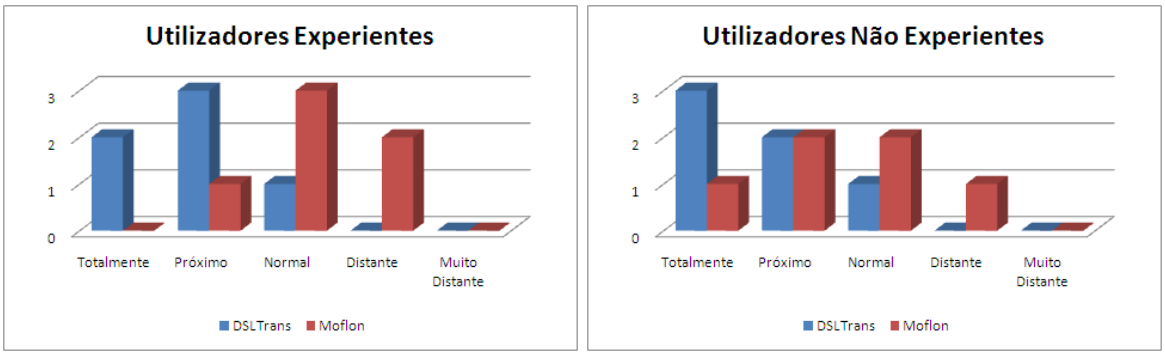

Figura 5.6: Resultado da questão EF2 - O resultado reflecte o que estava à espera?

5.1.6.5 Expressividade

A figura 5.7 indica que os utilizadores experientes não identificaram grandes diferenças na expressividade entre as duas ferramentas em análise. Um aspecto que pode indicar este resultado poderá ser a dimensão do cenário proposto. Embora suficiente para os utilizadores não experientes terem contacto com a linguagem, este não mostra todo o seu poder aos utilizadores habituados a expressões mais complicadas, não permitindo-lhes uma percepção total das suas potencialidades.

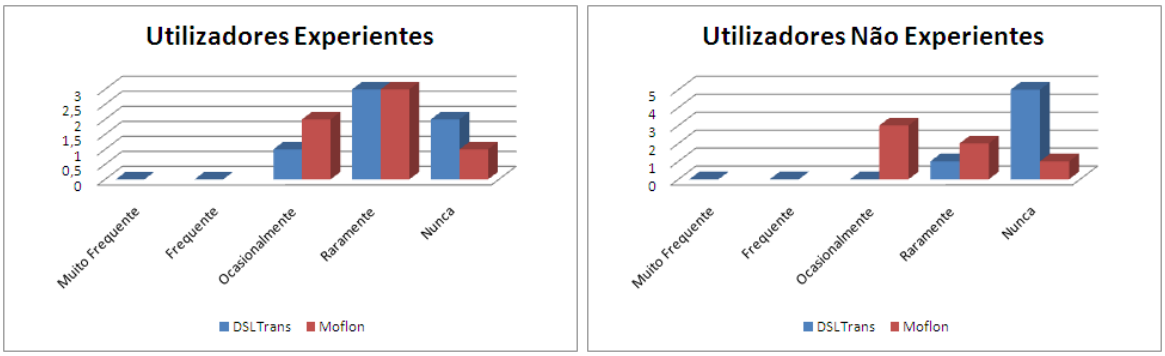

Figura 5.7: Resultados da questão EX2 - Com que frequência não conseguiu expressar o que pretendia?

5.1.7 Ameaças à validade

Aqui apresentamos possíveis ameaças que podem invalidar alguns dos resultados obtidos através do processo de avaliação apresentado.

5.1.7.1 Validade do Conteúdo

Uma possível ameaça a qualquer questionário prende-se com o facto dos utilizadores poderem deixar-se influenciar por outras respostas de utilizadores anteriores. Por esta razão no nosso teste, não era possível os utilizadores conhecerem os resultados anteriores do questionário. Além disso, os utilizadores foram seleccionados de diversos locais sendo improvável a troca de informação entre os mesmos. Como os resultados foram bastante semelhante, acreditamos que estes são suficientemente fiáveis para a validação da nossa ferramenta.

O questionário em si foi adaptado de uma dissertação [42] e por isso foi alvo de revisões por parte de pessoas com experiência na área.

5.1.7.2 Validade ao cenário

O cenário proposto neste teste experimental é bastante simples e dessa forma pode não ser o suficiente para que os utilizadores com experiência mais profunda em transformações de modelos tenham uma percepção real das capacidades da ferramenta e da sua real expressividade. No entanto, a escolha deste cenário deve-se ao facto de ser a primeira vez que uma parte significativa dos utilizadores tem contacto com transformações e desse modo um cenário demasiado complexo poderia influenciar a sua experiência. É também nosso entendimento que embora limitado, este cenário permite a ambos os grupos de utilizadores ter uma visão suficientemente apurada das capacidades da ferramenta de transformação desenvolvida.

5.1.7.3 Validade Interna

Uma questão que poderá influenciar as respostas dos utilizadores deve-se ao facto de todos eles serem pessoas do conhecimento da equipa de desenvolvimento da ferramenta e essa relação de proximidade poderia influenciar positivamente os resultados obtidos. Achamos porém, que os resultados apuraram-se de forma suficientemente independente até porque foi explicitamente explicado aos utilizadores que esta seria uma oportunidade de fornecer ideias úteis e funcionalidades que pretendiam ver satisfeitas numa ferramenta desta natureza.

5.2 Caso de Estudo

Como caso de estudo para a validação da nossa linguagem de transformação, utilizamos a definição de semântica de uma linguagem especificada em [39], em que foram definidas transformações ATL para definição da semântica da linguagem HALL[44] através da linguagem CO-OPN[45]. A escolha deste caso de estudo como base de validação da linguagem implementada, baseou-se na utilização de um caso real de semântica por transformação utilizado no contexto do projecto BATIC3S onde nos inserimos permitindo-nos ainda uma comparação real com outra linguagem de transformação bastante utilizada como seja o ATL. O trabalho utilizado como fonte de validação está, assim como esta linguagem de transformação, contido no projecto

BATIC3S aproveitando também a componente de validação deste trabalho como contribuição para o projecto em que se insere.

A utilização da ferramenta ATL como comparação neste caso de estudo assenta no facto de ser uma ferramenta com uma base de utilização bastante elevada, com um nível de expressividade bastante completo e sobretudo devido ao facto de existir dentro do projecto, um caso real de definição de semântica por transformação e com isso podermos avaliar o nível de expressividade da nossa ferramenta.

Para a compreensão completa deste caso de estudo é necessário um conhecimento prévio das linguagens HALL e CO-OPN que pode ser encontrado em [44] e [45] respectivamente.

5.2.1 Regras de Transformação

Apresentamos nesta secção alguns exemplo das regras de transformação implementadas, de forma a observar o nível de complexidade utilizado na definição da semântica da linguagem HALL. As figuras 5.8, 5.9, 5.10 e 5.11 apresentam uma fase inicial da definição semântica da linguagem HALL, com base nas regras definidas em [39]. Para uma visualização de todas as regras de transformação deste caso de estudo e devido à sua extensão é necessária a consulta do anexo B.

5.2.2 Resultados

Com este caso de estudo conseguimos demonstrar a boa expressividade da linguagem desenvolvida. O nível expressivo da linguagem mostrou-se suficiente num caso real de definição de semântica por transformação, provando que neste caso a nossa linguagem é tão expressiva quanto o ATL, embora isto possa não acontecer para outros domínios não estudados. De qualquer forma, acreditamos que a linguagem apresenta expressividade suficiente para não estar necessariamente limitada à definição de semântica de linguagens.

Este caso de estudo, permitiu ainda verificar que a mesma transformação foi definida em menos de metade das regras de transformação. A definição ATL descrita em [39] utilizou 89 regras ao passo que a definição em DSLTranslator utilizou apenas 41 para definir a mesma transformação. Isto permite-nos concluir que embora mantendo o mesmo nível de expressividade, neste caso, o número de regras necessárias é bastante inferior, o que tem impacto na produtividade uma vez que pode facilitar a análise do próprio modelo de transformação, nomeadamente por pessoas não directamente envolvidas no seu desenvolvimento.

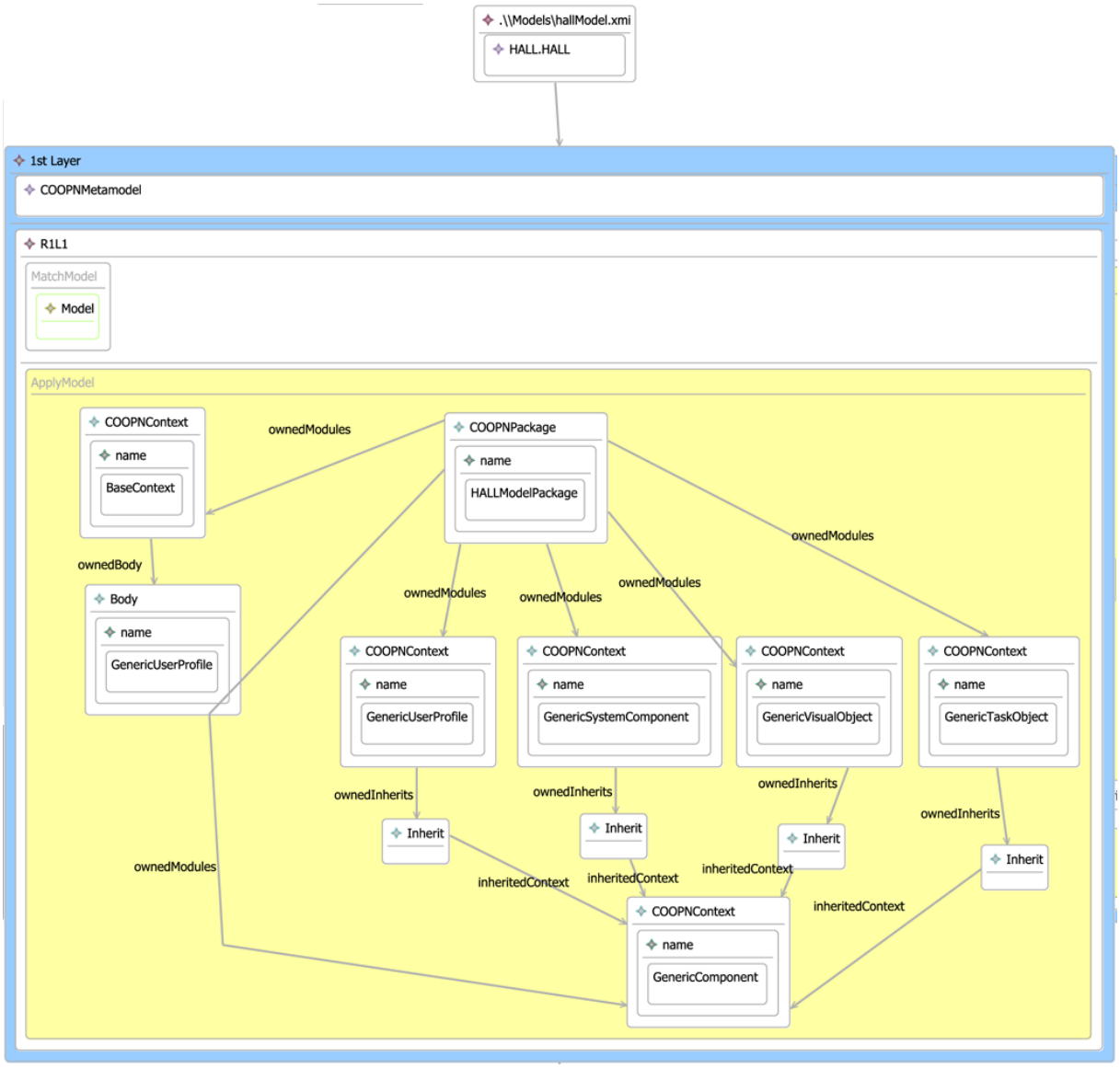

Figura 5.8: Primeira camada de regras de transformação entre a linguagem HALL e CO-OPN

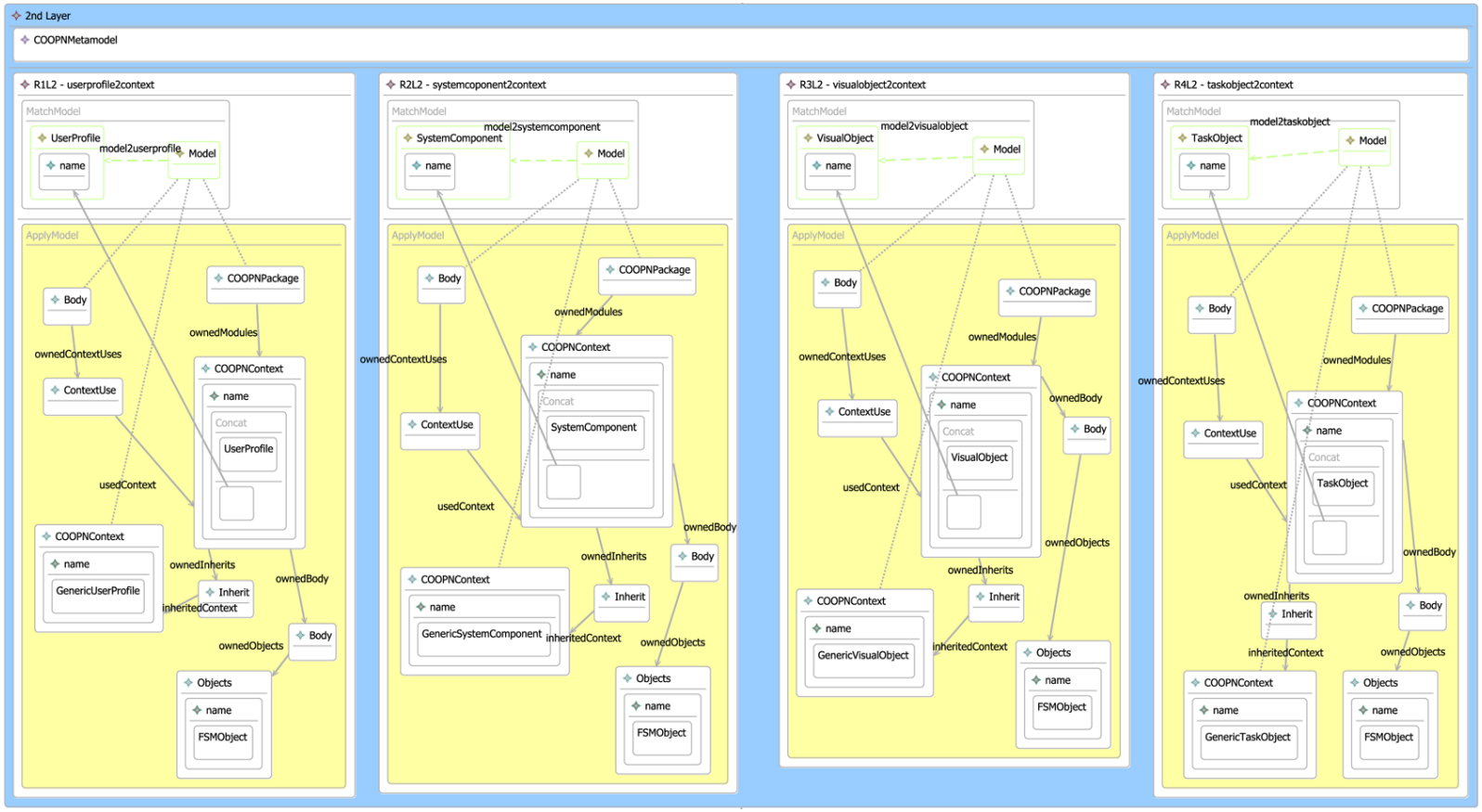

Figura 5.9: Segunda camada de regras de transformação entre a linguagem HALL e CO-OPN

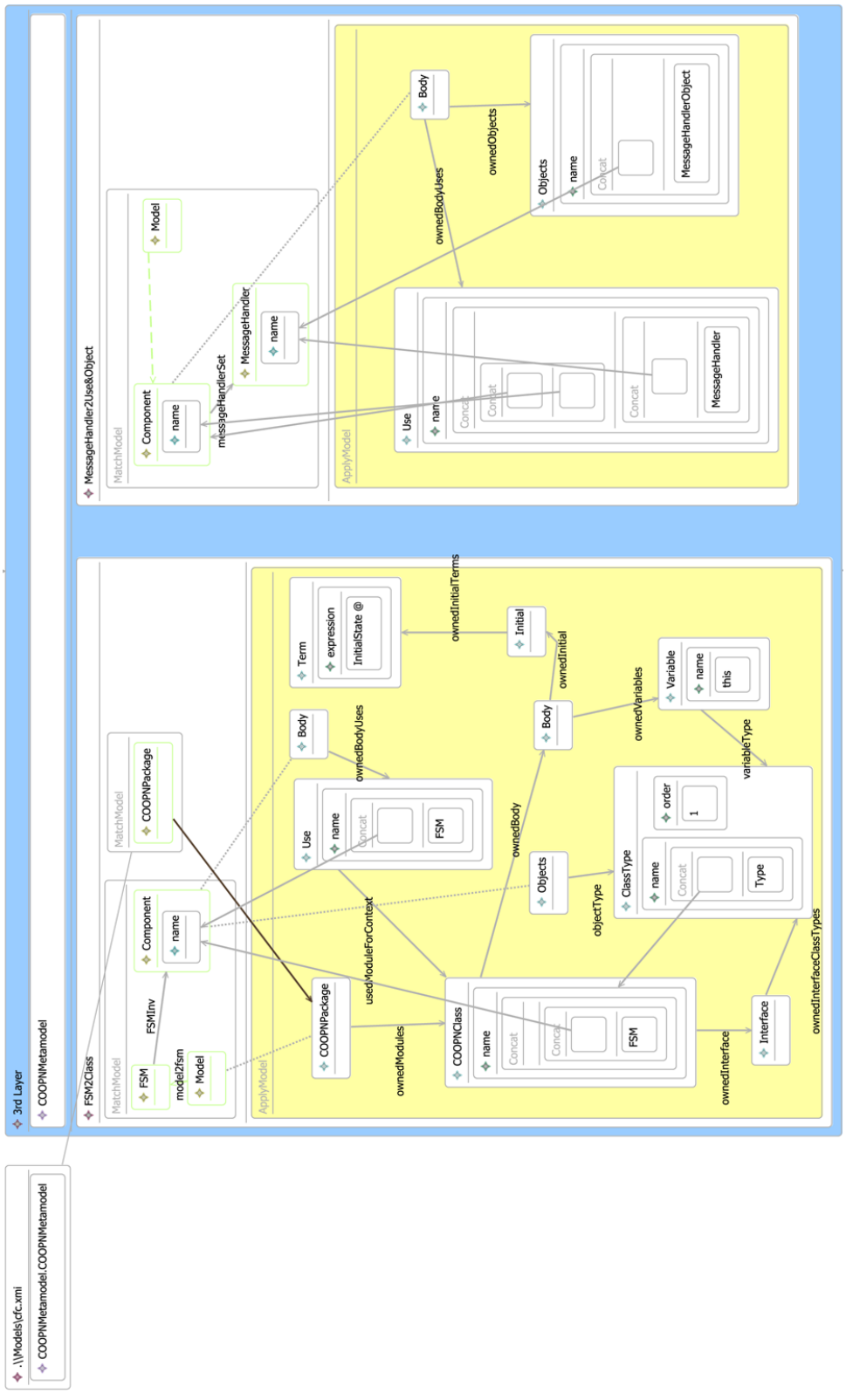

Figura 5.10: Terceira camada de regras de transformação entre a linguagem HALL e CO-OPN

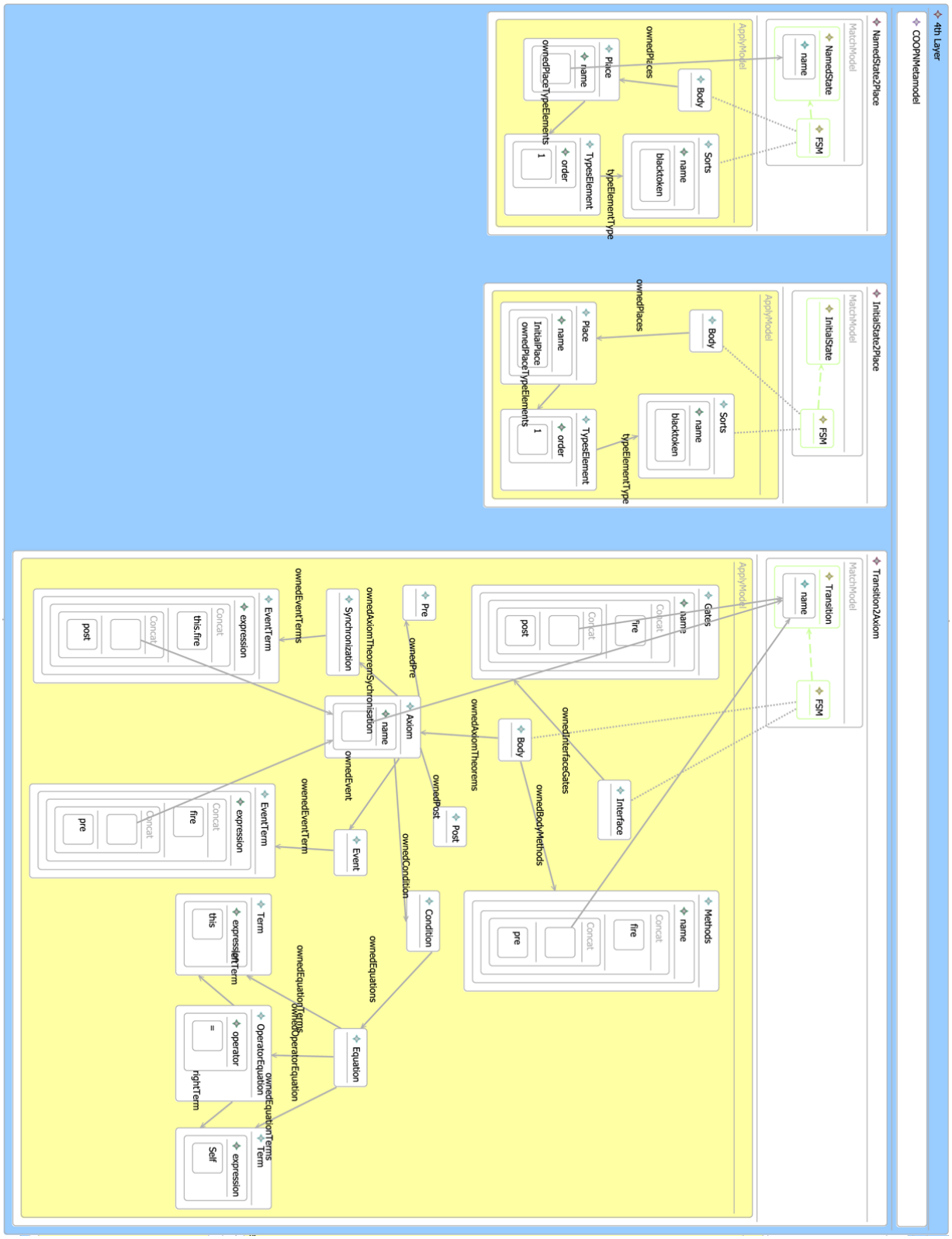

Figura 5.11: Quarta camada de regras de transformação entre a linguagem HALL e CO-OPN

6

Conclusões e Trabalho Futuro

Após completar esta dissertação de mestrado analisamos então o seu desenvolvimento, deixando ainda algumas sugestões sobre o trabalho que poderá ser feito na evolução da própria linguagem.

6.1 Conclusões

Em geral o trabalho desenvolvido no âmbito desta dissertação de mestrado, mostrou-se bastante satisfatório como se demonstrou nas secções anteriores pois foi desenvolvida uma ferramenta usável e que provou ter a expressividade necessária para o contexto em que foi desenvolvida, precisamente para a definição de semântica por transformação para linguagens de domínio específico sendo assim uma mais valia para o projecto onde se insere, o projecto BATIC3S.

Começamos esta dissertação através de uma análise às ferramentas de transformação mais utilizadas actualmente para que, de forma esclarecida, podermos avaliar quais as características a utilizar numa nova ferramenta de transformação e quais os pontos em que poderíamos marcar efectivamente a diferença.

Através do desenvolvimento de um novo motor de transformações, construímos a base do que se pretende que seja uma ferramenta de transformação de futuro, simples e eficiente. Todo o desenvolvimento teve por base a integração de novos utilizadores na definição de transformações de modelos e com isso contribuir também para a evolução do próprio desenvolvimento orientado a modelos.

Mostramos através de avaliações consistentes que a ferramenta mantém um equilíbrio eficaz entre expressividade e usabilidade de forma a conseguir ter como utilizadores alvo, não só pessoas com uma experiência muito grande em transformações de modelos, mas também novos engenheiros de linguagens sem experiência em nenhuma ferramenta, pois a curva de aprendizagem é consideravelmente inferior às ferramentas mais utilizadas. Desta forma evitamos que um engenheiro de linguagens tenha necessariamente que se tornar ele próprio um especialista de uma qualquer ferramenta de transformação, ficando assim livre para a sua actividade principal, ou seja, a criação de novas linguagens.

6.2 Trabalho Futuro

Como sugestões de continuação do trabalho desenvolvido nesta dissertação poderemos sugerir vários caminhos a seguir de forma a potenciar esta linguagem e respectiva ferramenta.

Um problema que pode surgir na utilização desta ferramenta tem precisamente a ver com o facto da performance do motor de transformação actualmente implementado. A transformação pode ficar comprometida com a utilização de modelos de entrada com um tamanho considerável, o que pode acontecer sobretudo devido à utilização de outras ferramentas de transformação de texto para modelos (*text-to-models*). Estas ferramentas automatizam o processo de criação de modelos e assim possibilitam a utilização de modelos bastante grandes como, por exemplo, na criação automática de um projecto Java para um modelo de entrada. Neste caso, seria interessante tornar o motor de transformação mais leve e rápido de forma a poder lidar com modelos de complexidades superiores sem prejuízo para o utilizador, nomeadamente na geração de soluções através do motor Prolog, pois consideramos ser este o causador principal da perda de performance da ferramenta. Deixamos como sugestão o artigo [46] para a continuação do trabalho nesta direcção.

Outro dos aspectos que gostaríamos de melhorar é a direccionalidade das transformações. Converter esta linguagem e respectiva ferramenta numa linguagem de transformação bidireccional traria uma maior base de utilização dada a possibilidade de utilização da mesma para fins, por exemplo, de sincronização de modelos, deixando de ser necessário a definição de regras de transformação inversas. Esta funcionalidade aproximar-nos-ia do standard QVT. Embora sem qualquer raciocínio específico neste caso julgamos que os conceitos presentes na linguagem podem permitir esta situação mesmo que as *Triple Graph Grammars* sejam a escolha natural para este caso.

Questionário completo utilizado na avaliação experimental da ferramenta

ID	Question
	Domain Experts Background
B1	Did you have software programming skills? (Yes, No) How do you classify yourself? (Experienced, Average, Beginner, Inexperienced) How many years do you program?
B2	How often did you use a DSL modelling tool? (Very often, Often, Sometimes, Seldom, Never) How long have you used it? Have you enjoyed it? What for?
	Learnability
L1	How easy did you find learning the concepts? (Very easy, Easy, Normal, Difficult, Very difficult)
L2	How useful have been the provided examples? (Very Good, Good, Satisfactory, Poor, Very Poor)
L3	How often did you find necessity to consult the documentation? (Very often, Often, Sometimes, Seldom, Never)
L4	How often did you perform questions to the supervisor? (Very often, Sometimes, Seldom, Never)
	Familiarity
F1	How do you identify the symbols representing the concepts? (Very Good, Good, Satisfactory, Bad, Very Bad) Which one did you find inadequate?

F2	How do you identify the text representing the concepts? (Very Good, Good, Satisfactory, Bad, Very Bad) Which one did you find inadequate?
F3	How often did you find committing errors due to symbols similarity? (Very often, Often, Sometimes, Seldom, Never)
F4	How often did you find committing errors due to ambiguous vocabulary? (Very often, Often, Sometimes, Seldom, Never)
	Ease of Use
U1	What do you think of the DSL tool? (Very Good, Good, Satisfactory, Bad, Very Bad)
U2	How confident did you feel during scenario execution? (Very confident, Confident, Normal, Insecure, Very insecure)
U3	How often did you find trapped or confused during the scenario? (Very often, Often, Sometimes, Seldom, Never)
U4	What did you feel about performing changes? (Very easy, Easy, Normal, Difficult, Very difficult)
U5	How mentally demanding was the scenario? (Very Difficult, Difficult, Normal, Simple, Very Simple) What did you feel more difficult to reason/perform?
	Effectiveness
EF1	How do you feel about the correctness of the performed scenario? (Very Good, Good, Satisfactory, Poor, Very Poor)
EF2	The outcome reflects what you were expecting? (Yes, No)
	Expressiveness
EX1	How compact did you find the accomplished scenario? (Very compact, Compact, Expected, Extensive, Very Extensive)
EX2	How often did you feel unable to express what you intended? (Very often, Often, Sometimes, Seldom, Never) What?
	General Impressions
G1	What is your overall appreciation of the DSL? (Very Good, Good, Satisfactory, Poor, Very Poor)
G2	What changes or additions do you propose to the model?
G3	Do you feel the new system as a value-added compared to the previous one? (Yes, No) Why?
G4	Do you feel more productive than with the previous system? (Yes, No) Why?

Definição da transformação entre HALL e CO-OPN

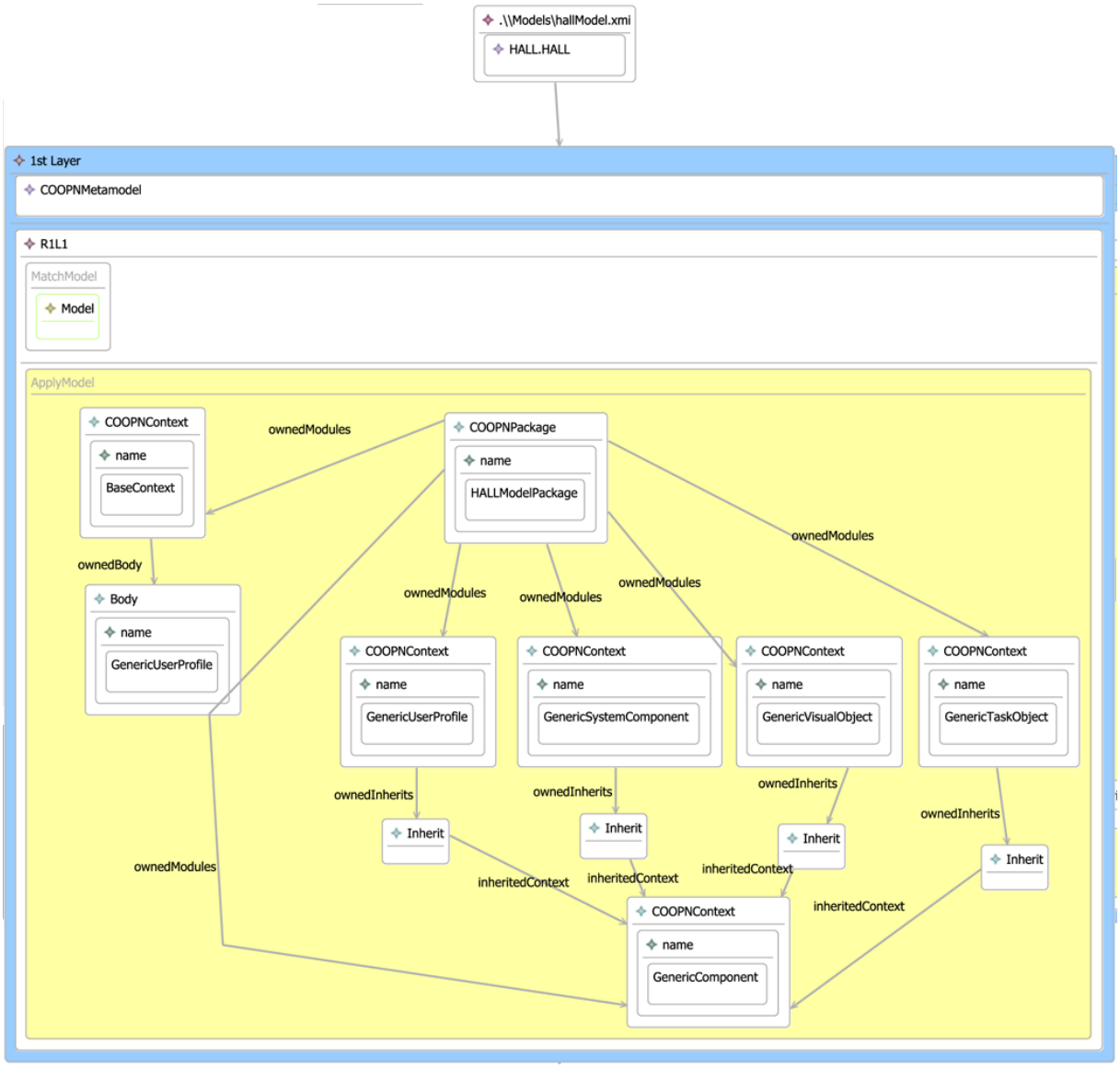

Figura B.1: Primeira camada de regras de transformação entre a linguagem HALL e CO-OPN

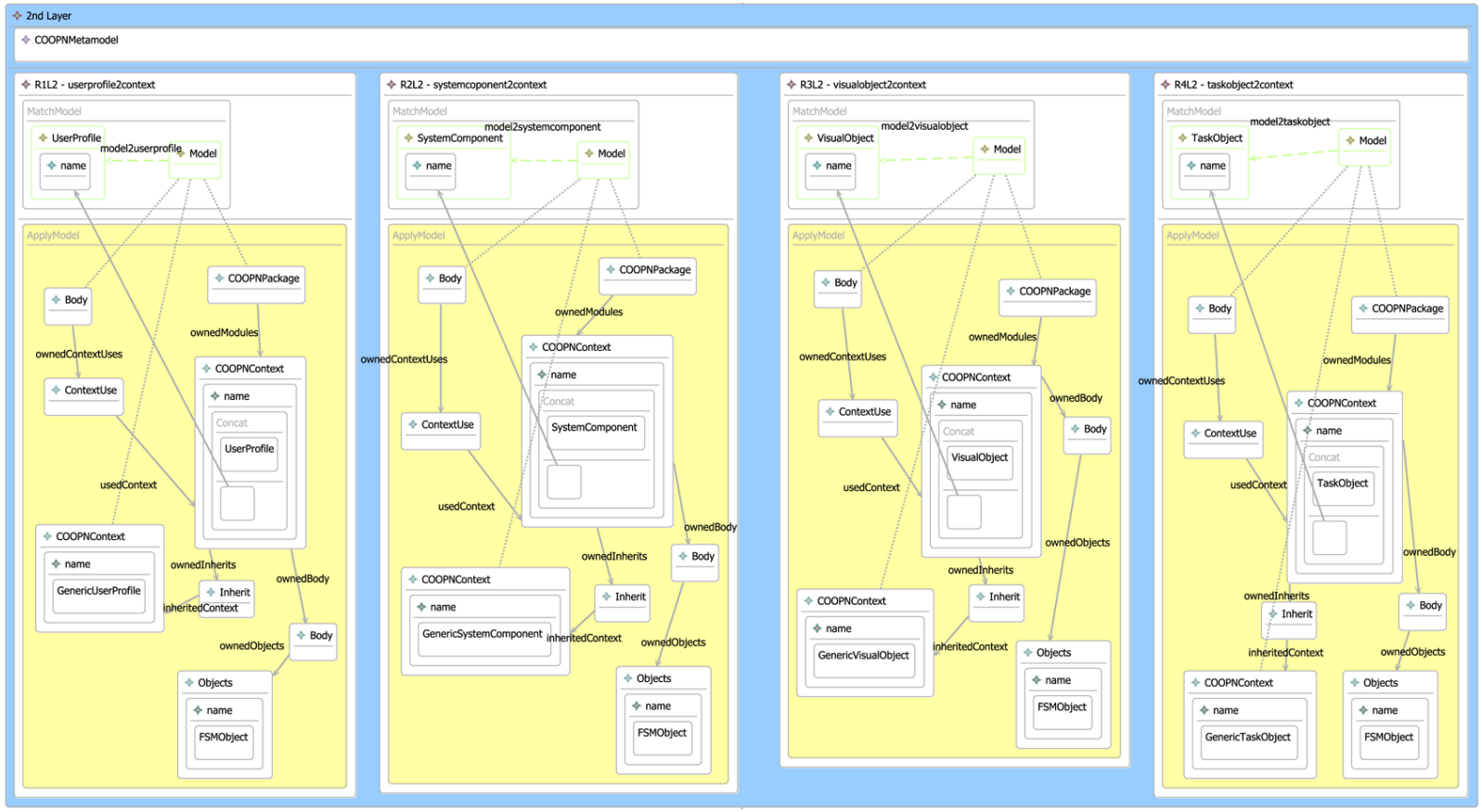

Figura B.2: Segunda camada de regras de transformação entre a linguagem HALL e CO-OPN

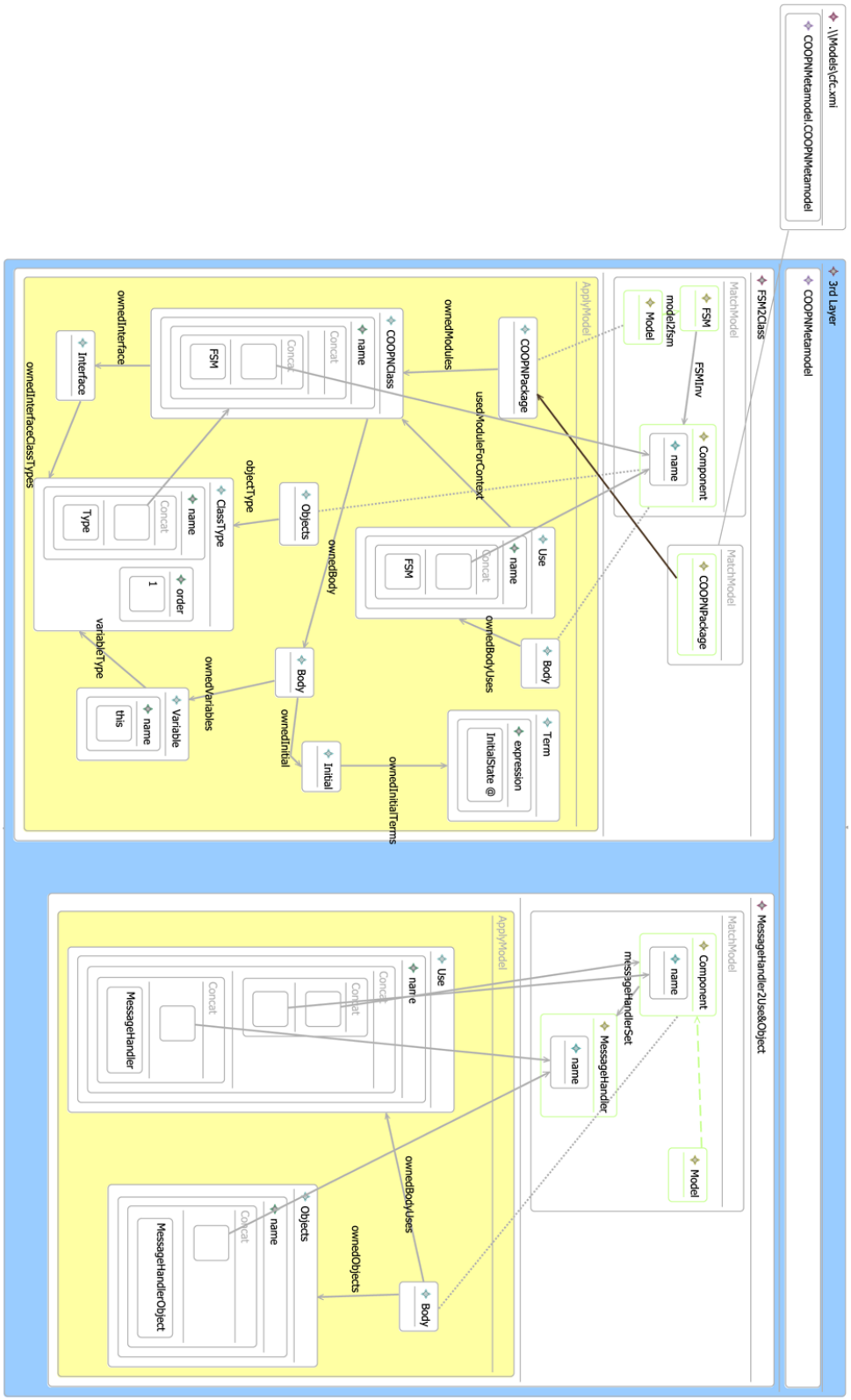

Figura B.3: Terceira camada de regras de transformação entre a linguagem HALL e CO-OPN

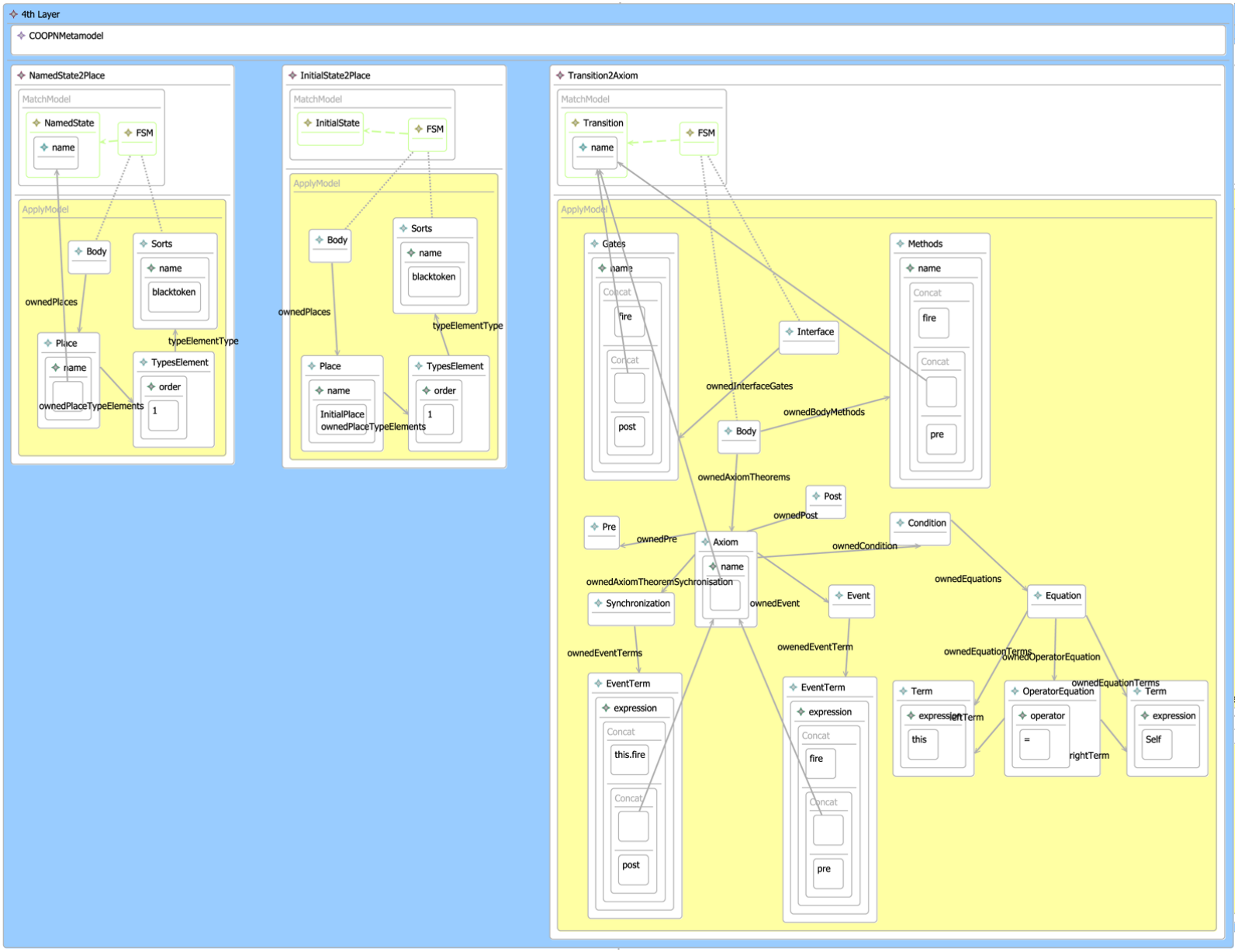

Figura B.4: Quarta camada de regras de transformação entre a linguagem HALL e CO-OPN

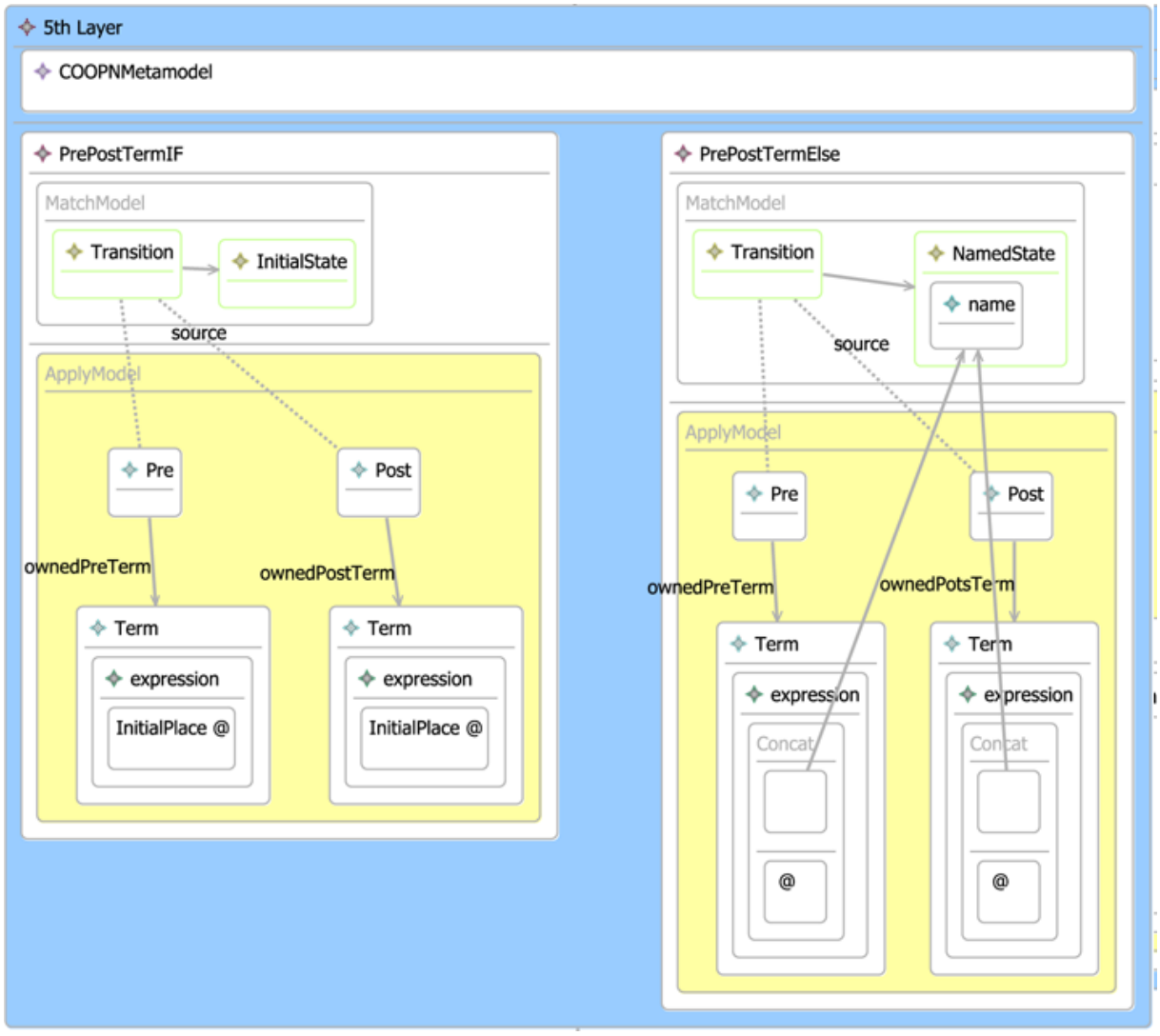

Figura B.5: Quinta camada de regras de transformação entre a linguagem HALL e CO-OPN

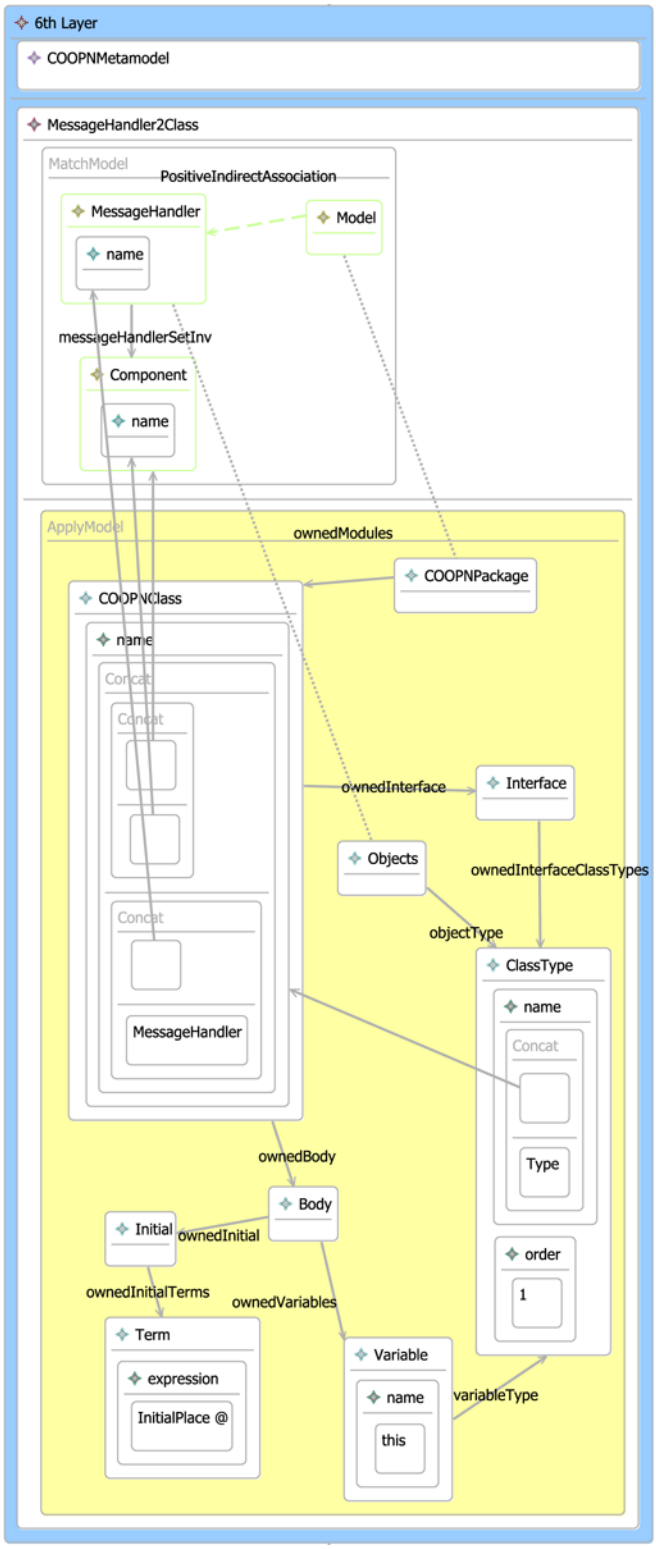

Figura B.6: Sexta camada de regras de transformação entre a linguagem HALL e CO-OPN

Figura B.7: Sétima camada de regras de transformação entre a linguagem HALL e CO-OPN

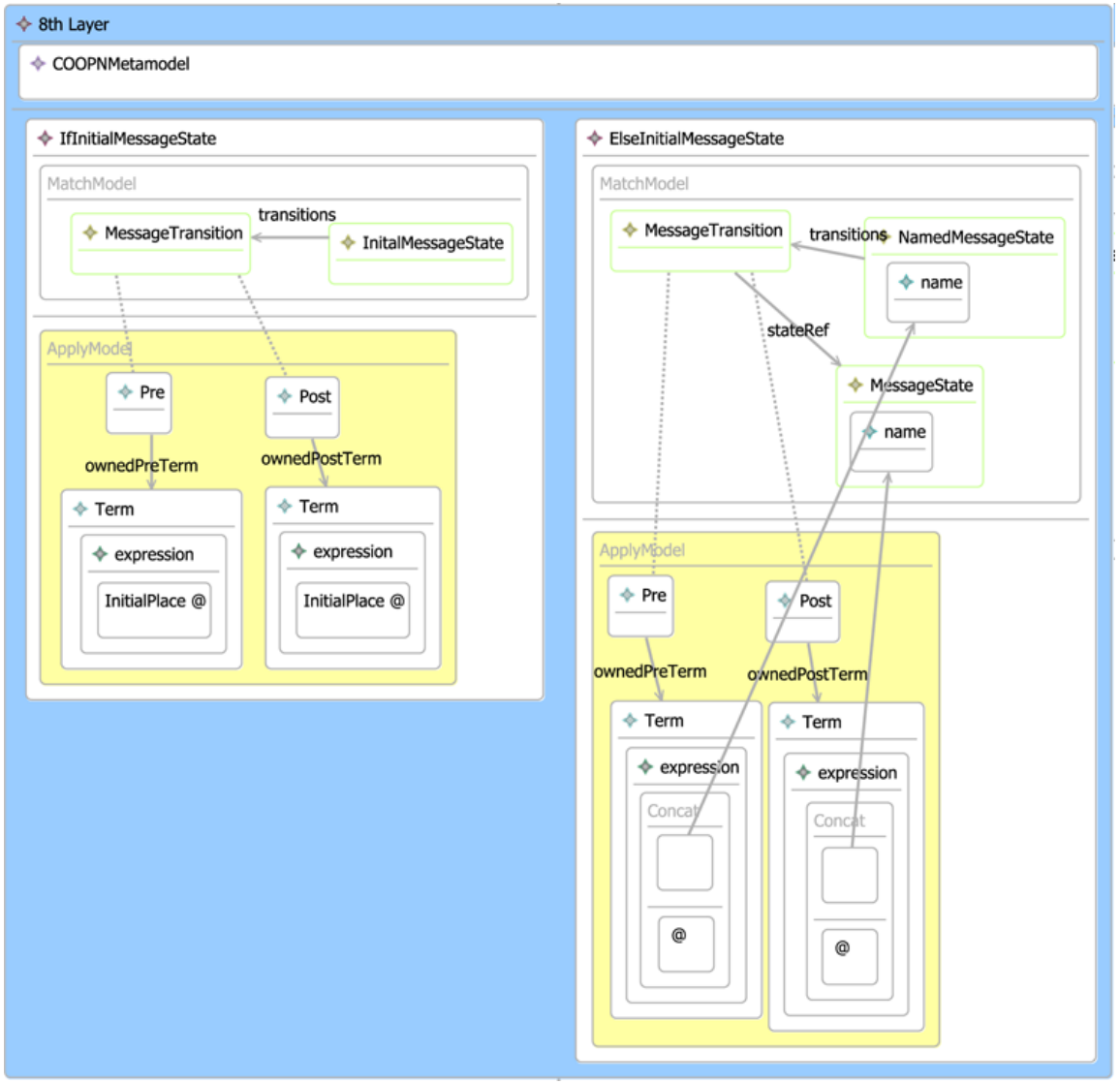

Figura B.8: Oitava camada de regras de transformação entre a linguagem HALL e CO-OPN

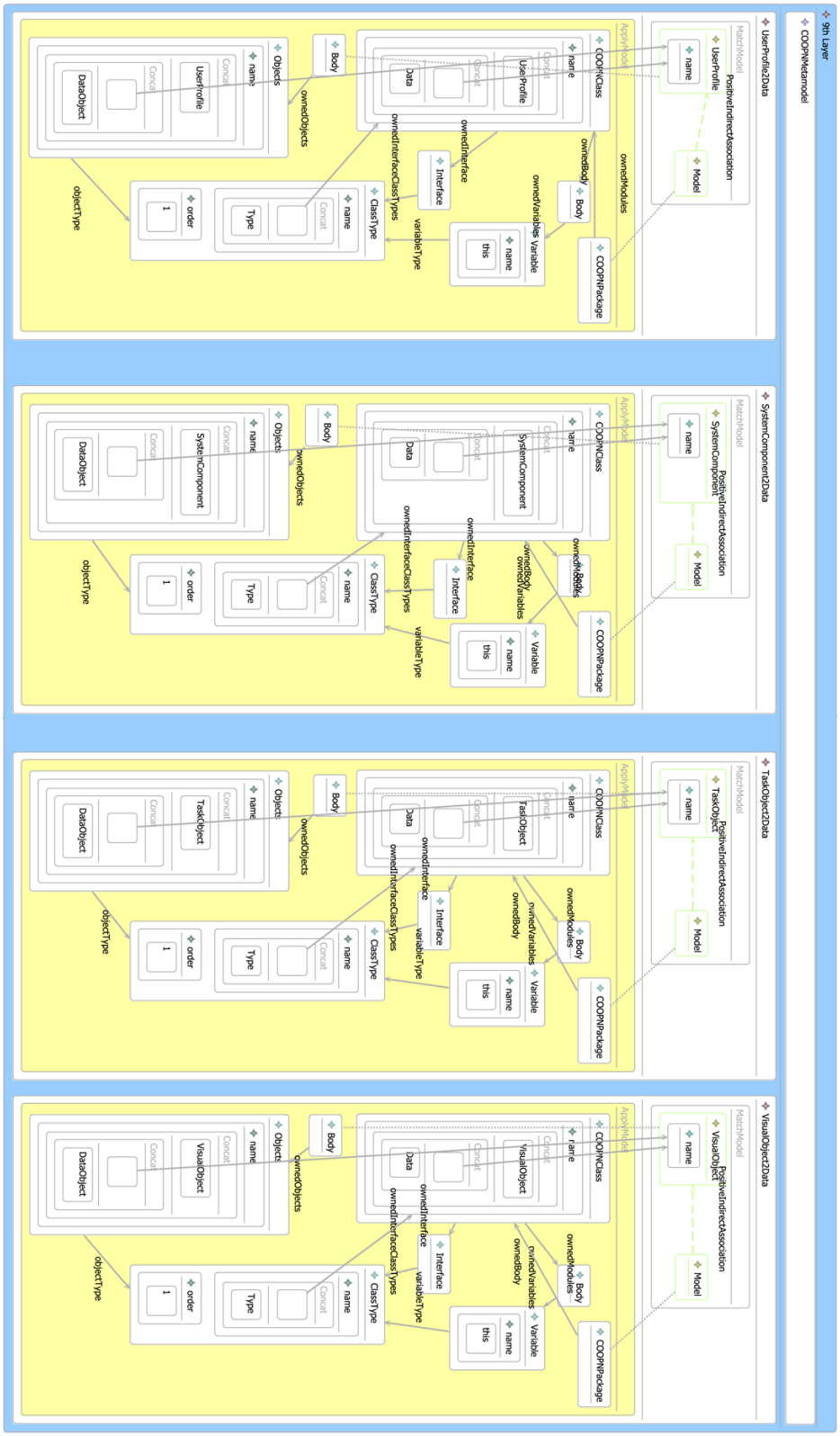

Figura B.9: Nona camada de regras de transformação entre a linguagem HALL e CO-OPN

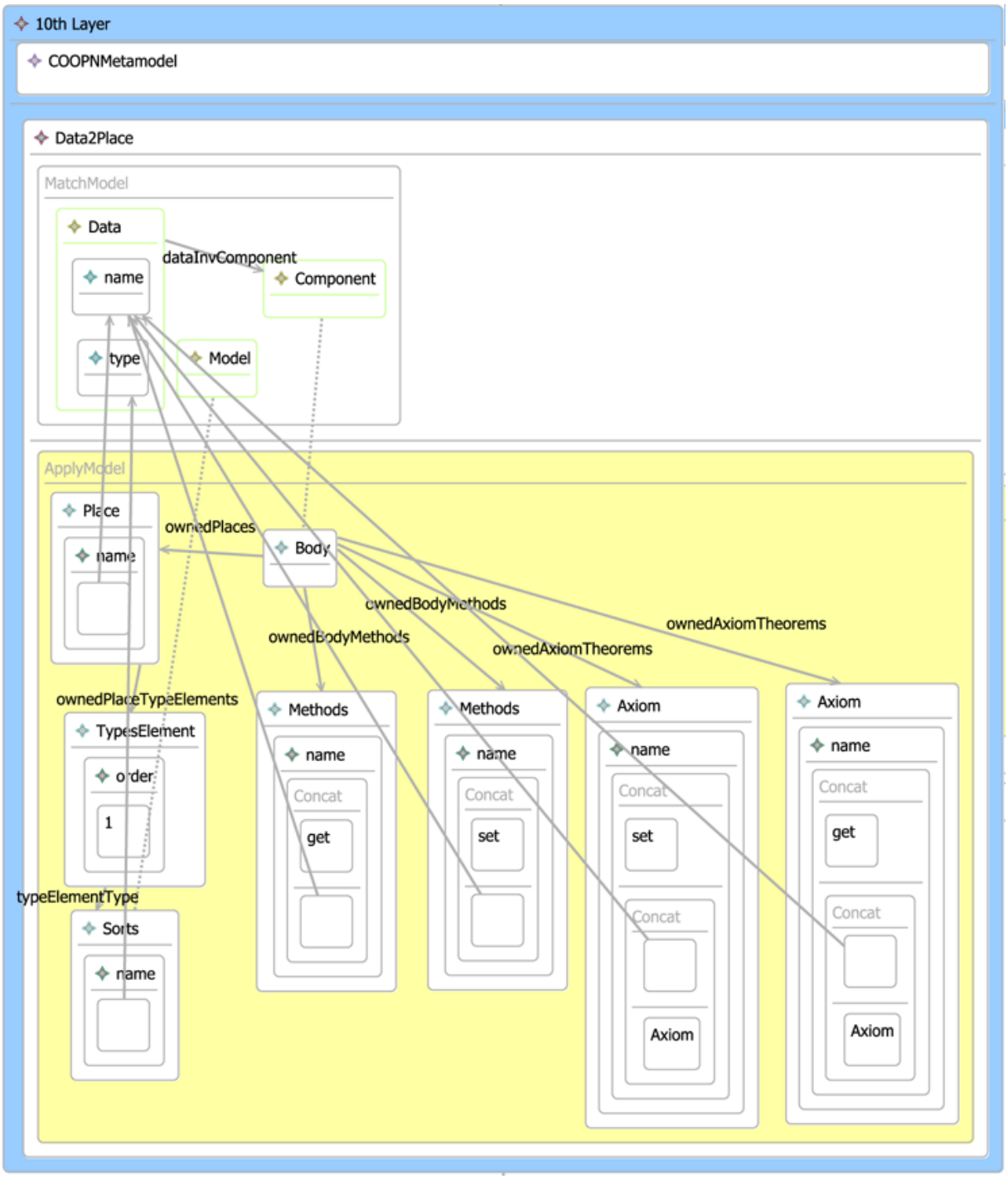

Figura B.10: Décima camada de regras de transformação entre a linguagem HALL e CO-OPN

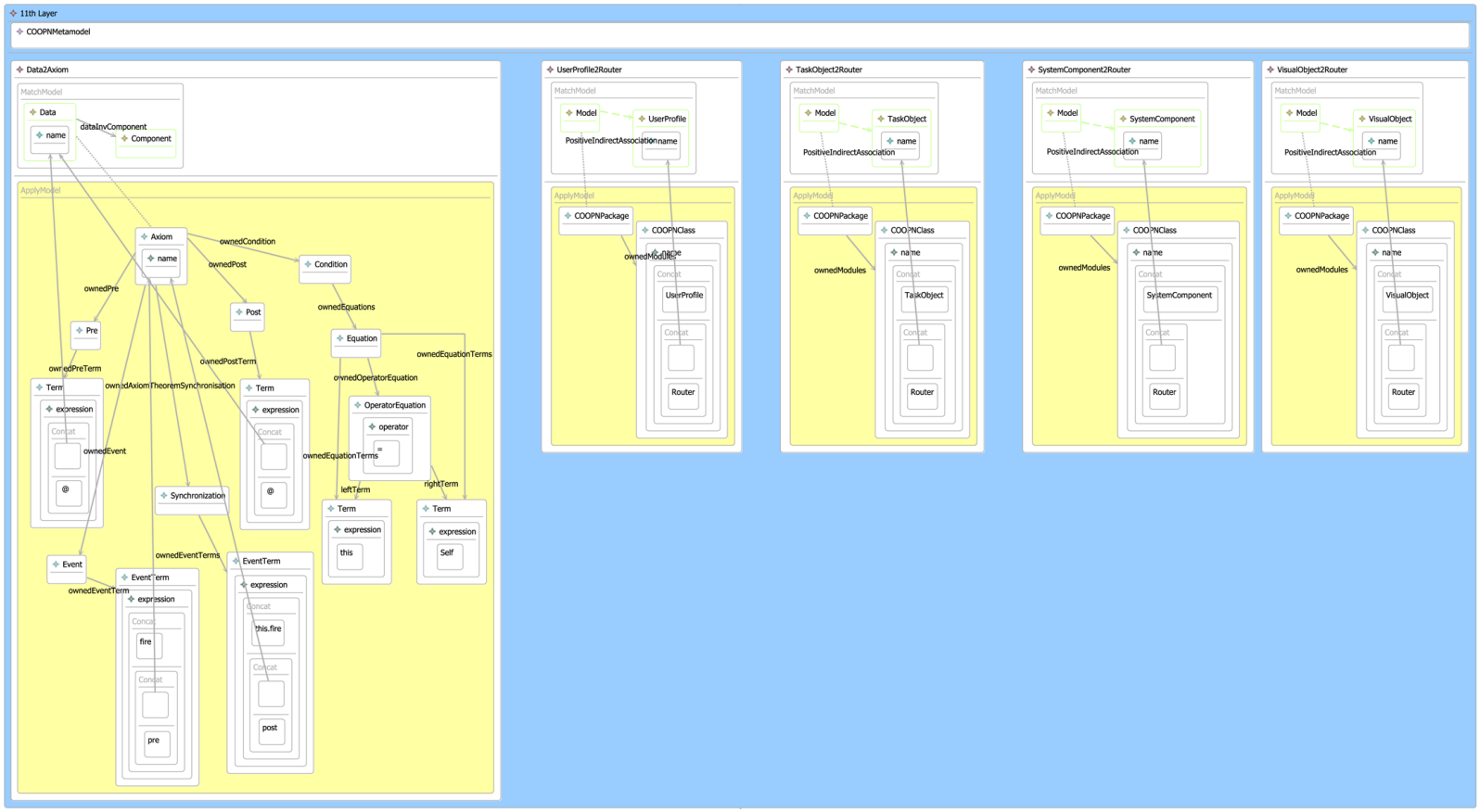

Figura B.11: Décima primeira camada de regras de transformação entre a linguagem HALL e CO-OPN

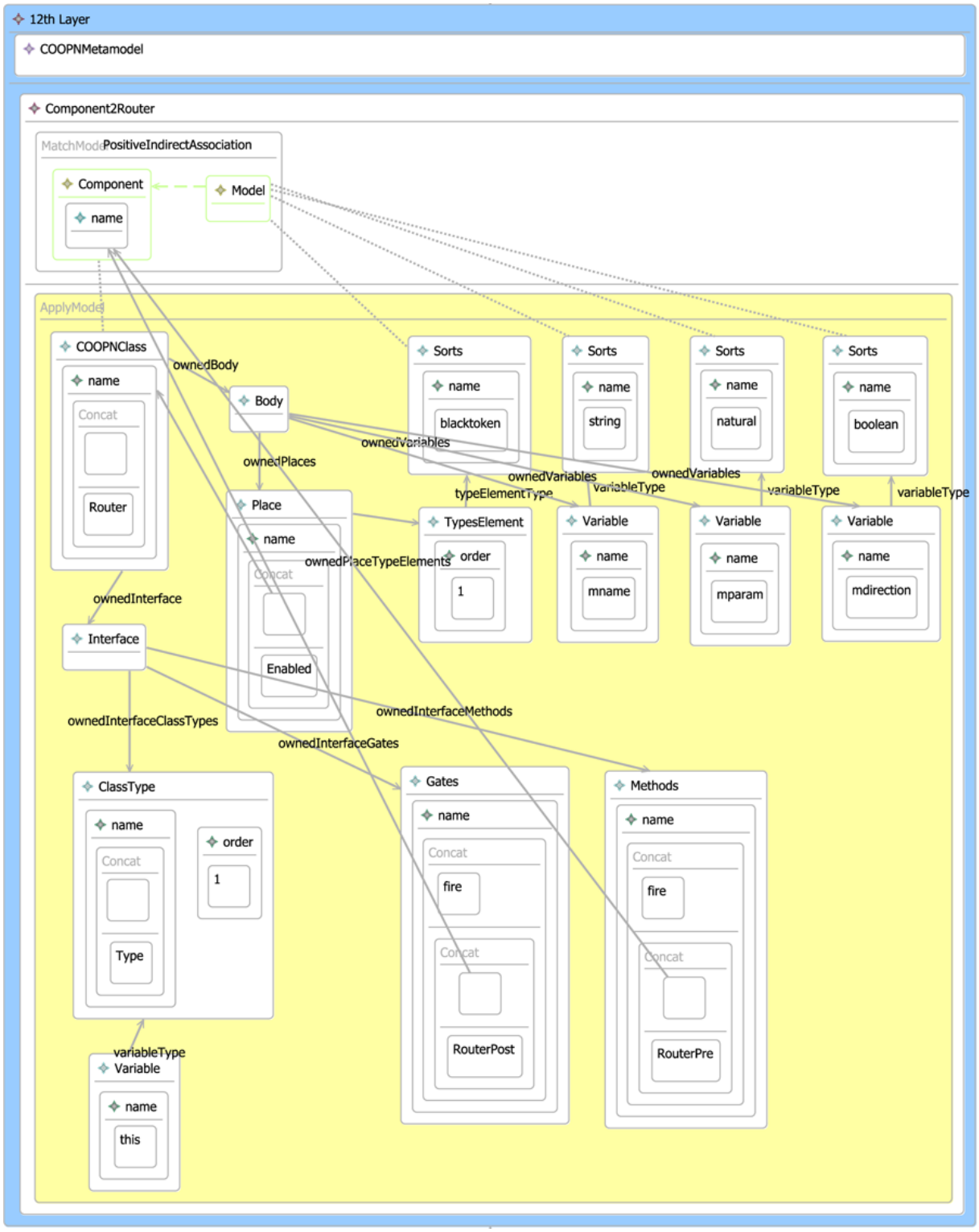

Figura B.12: Décima segunda camada de regras de transformação entre a linguagem HALL e CO-OPN

Figura B.13: Décima terceira camada de regras de transformação entre a linguagem HALL e CO-OPN

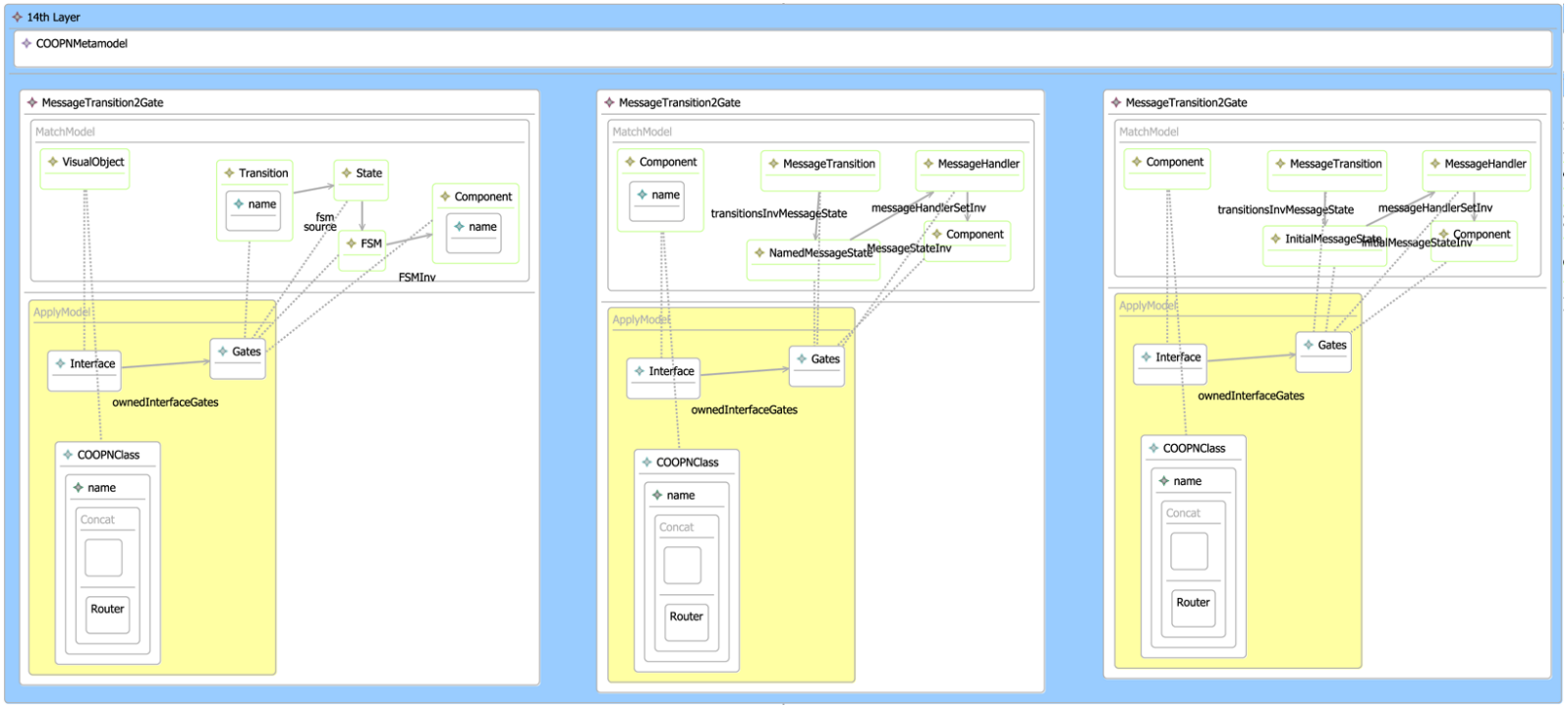

Figura B.14: Décima quarta camada de regras de transformação entre a linguagem HALL e CO-OPN

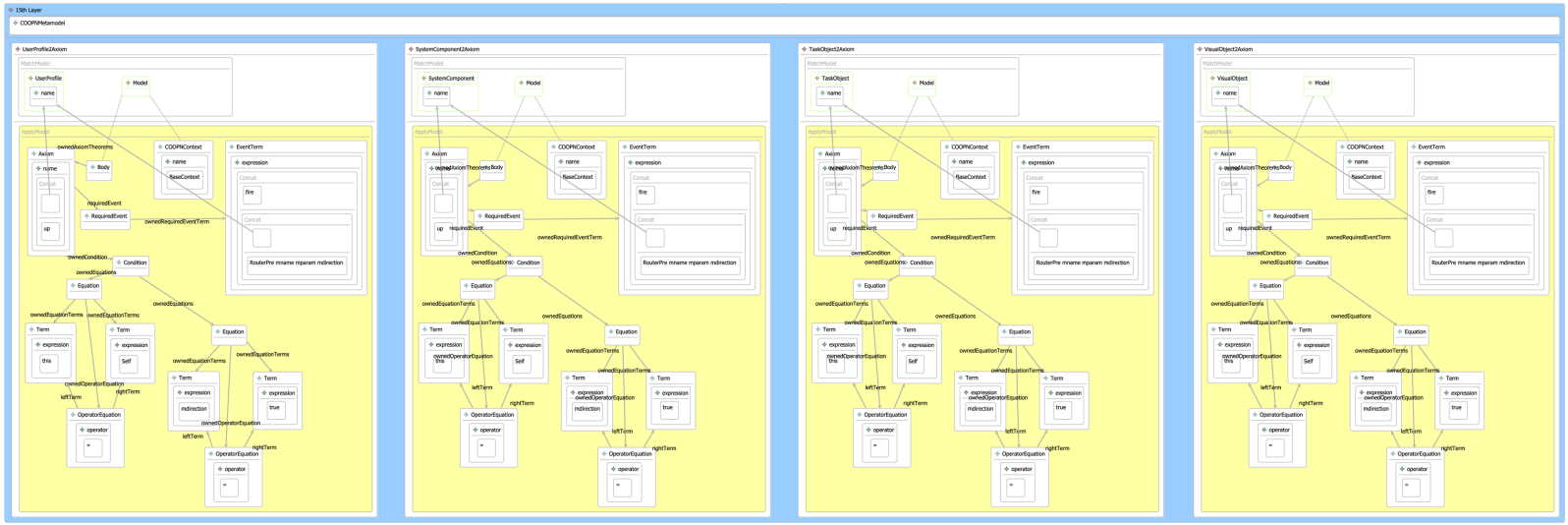

Figura B.15: Décima quinta camada de regras de transformação entre a linguagem HALL e CO-OPN

Bibliografia

[1] Matteo Risoldi and Vasco Amaral. Towards a Formal, Model-Based Framework for Control Systems Interaction Prototyping. In Didier Buchs Nicolas Guelfi, editor, *RISE 2006 International Workshop on Rapid Integration of Software Engineering techniques 13-15 September*, number 4401 in Springer Verlag Lecture Notes in Computer Science 4401 (LNCS) Series 2007, pages 144–159. Springer-Verlag, 2007.

[2] Jean Bézivin. On the Unification Power of Models. *Software and System Modeling*, 4(2):171–188, May 2005.

[3] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, 2006.

[4] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework: A Developer's Guide*. Addison Wesley, 2003.

[5] Technischen Universität Berlin, Berlin. *Tiger EMF Model Transformation Framework (EMT) User Manual*, 1.2.0 edition, April 2007. http://user.cs.tu-berlin.de/ emf-trans/papers/userdoc.pdf.

[6] Aditya Agrawal, Gabor Karsai, and Feng Shi. Graph Transformations on Domain-Specific Models. Technical report, Institute for Software Integrated Systems, Vanderbilt University, 2003.

[7] András Balogh and Dániel Varró. Advanced model transformation language constructs in the VIATRA2 framework. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1280–1287, New York, NY, USA, 2006. ACM.

[8] Radomil Dvorak. Model Transformation with Operational QVT. In *EclipseCon2008 - Presentation Slides*, 2008.

[9] Andy Schürr. PROGRES, A Visual Language and Environment for PROgramming with Graph REwriting Systems. Technical report, Lehrstuhl für Informatik III, RWTH Aachen, 1994.

[10] Holger Giese, Sabine Glesner, Johannes Leitner, Wilhelm Schäfer, and Robert Wagner. Towards Verified Model Transformations. In David Hearnden, Jörn Guy Süß, Benoît Baudry, and Nicolas Rapin, editors, *Proc. of the 3rd International Workshop on Model Development, Validation and Verification (MoDeV²a), Genova, Italy*, pages 78–93. Le Commissariat à l'Energie Atomique - CEA, October 2006.

[11] Alexander Königs, Felix Klar, and Sebastian Rose. *MOFLON - TGG by Example, A guided tour through your First Triple Graph Grammar specification*, 1.4 edition, November 2009. http://www.moflon.org/documentation/tutorial/.

[12] Zoé Drey, Cyril Faucher, Franck Fleurey, Vincent Mahé, and Didier Vojtisek. *Kermeta language - Reference Manual*. Institut de Recherche en Informatique et Systèmes Aléatoires, France, April 2009.

[13] Pierre alain Muller, Franck Fleurey, Zoé Drey, Damien Pollet, and Frédéric Fondement. On executable meta-languages applied to model transformations. In *Model Transformations In Practice Workshop, Montego*, 2005.

[14] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *OOPSLA '03: Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[15] Ernest Lepore and Barry Loewer. Translational semantics. *Synthese*, 48(1):121–133, July 1981.

[16] Matteo Risoldi, Vasco Amaral, Bruno Barroca, Kaveh Bazargan, Didier Buchs, Fabian Cretton, Gilles Falquet, Anne Le Calvé, Stéphane Malandain, and Pierrick Zoss. A Language and a Methodology for Prototyping User Interfaces for Control Systems. *Human Machine Interaction - Research Results of the MMi Program*, 2009.

[17] The Object Management Group. *QVT Specification*, 04 2008. `http://www.omg.org/spec/QVT/1.0/`.

[18] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, March 2006.

[19] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *MoDELS Satellite Events*, pages 128–138, 2005.

[20] Stephen J. Mellor, Anthony N. Clark, and Takao Futagami. Guest Editors' Introduction: Model-Driven Development. *IEEE Software*, 20(5):14–18, 2003.

[21] Shane Sendall and Wojtek Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, 2003.

[22] The Object Management Group. *MOF Specification*, April 2002. `http://www.omg.org/technology/documents/formal/mof.htm`.

[23] The Object Management Group. *Unified Modeling Language: Infrastructure*, July 2005.

[24] Arturo Sánchez-Ruíz, Motoshi Saeki, Benoit Langlois, and Roberto Paiano. Domain-Specific Software Development Terminology: Do We All Speak the Same Language? In *7th OOPSLA Workshop on Domain-Specific Modeling*, 2007.

[25] Arie V. Van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[26] Steven Kelly and Juha-Pekka Tolvanen. *Enabling Full Code Generation*. John Wiley and Sons, 2008.

[27] Jean Bézivin, Fabian Büttner, Martin Gogolla, Frédéric Jouault, Ivan Kurtev, and Arne Lindow. Model Transformations? Transformation Models! In *MoDELS*, pages 440–453, 2006.

[28] Daniel Exertier, Benoit Langlois, and Xavier Leroux. PIM Definition and Description. In *First European Workshop Model-Driven Architecture with Emphasis on Industrial Applications*, March 2004.

[29] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[30] Heiko Dorr. *Efficient Graph Rewriting and Its Implementation*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1995.

[31] Lars Grunske, Leif Geiger, Albert Zündorf, N. Van Eetvelde, Pieter Van Gorp, and Daniel Varro. Using Graph Transformation for Practical Model Driven Software Engineering. *Model-driven Software Development*, pages 91–118, 2005.

[32] Gabriele Taentzer, Karsten Ehrig, Esther Guerra, Juan de Lara, Laszlo Lengyel, Tihamer Levendovszky, Ulrike Prange, Daniel Varro, and Szilvia Varro-Gyapay. Model Transformation by Graph Transformation: A Comparative Study. In *MTiP 2005, International Workshop on Model Transformations in Practice*, 2005.

[33] Dimitrios S. Kolovos, Louis M. Rose, Richard F. Paige, and Fiona A. C. Polack. Raising the level of abstraction in the development of gmf-based graphical model editors. In *MISE '09: Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering*, pages 13–19, Washington, DC, USA, 2009. IEEE Computer Society.

[34] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Precise Semantics of EMF Model Transformations by Graph Transformation. In *MODELS'08: Proceedings of ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems*, pages 53–67, Berlin, Germany, 2008. Springer.

[35] Dániel Varró and András Balogh. The model transformation language of the VIATRA2 framework. *Sci. Comput. Program.*, 68(3):187–207, 2007.

[36] U. Nickel, J. Niere, and A. Zundorf. The FUJABA Environment. *International Cconference On Software Engineering*, 2000.

[37] R. Wagner. Developing Model Transformations with Fujaba. *4th Int. Fujaba Days*, 2006.

[38] Carsten Amelunxen, Alexander Königs, Tobias Rötschke, and Andy Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *ECMDA-FA*, pages 361–375, 2006.

[39] Vasco Nuno da Silva de Sousa. Model Driven Development: Implementation of a Control Systems User Interfaces Specification Tool. Master's thesis, Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa, 2009.

[40] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models. *Software Process: Improvement and Practice*, 2005.

[41] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[42] Pedro Hugo do Nascimento Gabriel. Software Languages Engineering: Experimental Evaluation. Master's thesis, Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa, 2010.

[43] Jakob Nielsen and Thomas K. Landauer. A mathematical model of the finding of usability problems. In *CHI '93: Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, pages 206–213, New York, NY, USA, 1993. ACM.

[44] Bruno Barroca and Vasco Amaral. (H)ALL: a DSVL for designing user interfaces for Control Systems. In *Proceedings of the 5th Nordic Workshop on Model Driven Engineering NW-MoDE*, Ronneby, Sweden, August 2007. Blekinge Institute of Technology.

[45] Olivier Biberstein, Didier Buchs, and Nicolas Guelfi. Object-Oriented Nets with Algebraic Specifications: The CO-OPN/2 Formalism. In G. Agha, F. De Cindio, and G. Rozenberg, editors, *Advances in Petri Nets on Object-Orientation*, LNCS, pages 70–127. Springer-Verlag, 2001.

[46] Gustaf Neumann. A Simple Transformation from Prolog-written Metalevel Interpreters into Compilers and its Implementation. In *Proceedings of the First Russian Conference on Logic Programming*, pages 349–360, London, UK, 1992. Springer-Verlag.